MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

Analyse de classes Android

Bastin, Laura

Award date:
2020

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Analyse de classes Android

Laura BASTIN

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Table des matières

Résumé

Android étant le système d'exploitation pour mobiles le plus utilisé, il est crucial de pouvoir s'assurer, autant pour l'utilisateur que pour le développeur d'applications, que ces dernières sont fiables.

Une des techniques pour vérifier cette fiabilité est l'analyse statique (analyse de diverses caractéristiques du programme sans l'exécuter).

Dans ce travail il sera question, plus précisément, de la construction des graphes d'appels de ces applications (ici, à partir de leur *bytecode* Dalvik) qui peuvent aider à mettre en évidence des structures de code dangereuses (comme dans [6]).

Cependant, il n'existe pas d'outil de construction de graphes d'appels à l'heure actuelle qui inclut toutes les spécificités d'Android (*intents*, fichiers de *layout*, ...) dans ses graphes d'appels, ce qui est utile pour avoir un vision plus précise des appels entre méthodes (dont certains pourraient être cachés par ces spécificités d'Android).

Pour construire ces graphes d'appels, un outil qui existe déjà - Rundroid - sera exploité et amélioré.

Ils pourront être générés soit via l'algorithme RTA ou XTA (tels qu'exposés dans le papier de TIP et PALSBERG : [31]), soit via l'algorithme 0-CFA de Jean PRIVAT ([29]).

Ces graphes prennent en compte les points d'entrée multiples, les fichiers de *layout*, la réflexion et les *intents*, ce qui permet une analyse plus détaillée et plus précise de l'application.

1 Introduction

Le but de ce travail est d'améliorer un outil d'analyse statique d'applications Android (Rundroid [1]) en y ajoutant la possibilité de construire des graphes d'appels d'applications Android en ayant le choix de l'algorithme : RTA, XTA ou 0-CFA. Aussi, ces graphes d'appels tiendront compte des spécificités d'Android comme les points d'entrées multiples, les fichiers de *layout*, la réflexion et les *intents*.

Il faut préablement préciser que le but d'un graphe d'appels est de visualiser les appels entre les différents sous-programmes d'un programme selon [29]. Dans le contexte de ce travail, les sous-programmes seront des méthodes.

L'algorithme RTA (ou *Rapid Type Analysis*) est un algorithme de construction de graphes d'appels pour les programmes Java qui utilise un ensemble conservant les classes instanciées du programme et un ensemble contenant les méthodes atteignables de ce même programme.

- Au tout début de l'algorithme, la méthode **public static void main (String[])** est toujours ajoutée à l'ensemble des méthodes atteignables (qui était au préalable vide). L'ensemble des classes instanciées reste vide.
- À chaque fois qu'une occurrence de **new** est rencontrée dans le programme dans une méthode atteignable, la classe instanciée par le **new** est ajoutée à l'ensemble des classes instanciées.
- À chaque appel de méthode dans une méthode atteignable, si la classe de la méthode appelée est présente dans l'ensemble des classes instanciées, une arête est ajoutée entre la méthode appelante et la méthode appelée **m()** de cette classe. Une arête est également ajoutée entre la méthode appelante et cette méthode **m()** des sous-classes de cette classe qui figurent dans l'ensemble des classes instanciées.

L'algorithme peut être formalisé de cette manière :

```
soit R = {} et S ={}, respectivement, ensemble des méthodes
atteignables et ensemble des classes instanciées.
A -> B signifie ajout de A dans B.
A ->> B signifie arête ajoutée entre A et B.

public static void main(String[]) -> R

pour chaque m() dans R :
        pour chaque new C() dans m() :
                C -> S
        pour chaque c.n() dans m() :
                si type(c) dans S :
                        type(c).n() -> R
                        m() ->> type(c).n()
                pour chaque sous-type(c) dans S :
                        si n() dans sous-type(c) :
                                sous-type(c).n() -> R
                                m() ->> sous-type(c).n()
```

L'algorithme XTA (ou *Separate Type Analysis*) étend l'algorithme RTA de la manière suivante : Au lieu d'avoir un ensemble contenant les classes instanciées

1. https ://bitbucket.org/etiennepayet/rundroid

de tout le programme, XTA possède un ensemble pour chaque méthode et un ensemble pour chaque champ.

- À chaque fois qu'une occurrence de **new** est rencontrée dans le programme dans une méthode atteignable, la classe instanciée par cette instruction **new** est ajoutée l'ensemble de cette méthode.
- À chaque fois qu'une lecture d'un champ survient dans une méthode atteignable, le contenu de l'ensemble de ce champ est ajouté à l'ensemble de la méthode atteignable.
- À chaque fois qu'une écriture dans un champ survient dans une méthode atteignable, le type et les sous-types de ce champ déjà dans l'ensemble de la méthode atteignable sont ajoutés à l'ensemble du champ.
- À chaque appel de méthode dans une méthode atteignable, si la classe de la méthode appelée est présente dans l'ensemble de la méthode appelante, une arête est ajoutée entre la méthode appelante et la méthode appelée **m()** de cette classe. Une arête est également ajoutée entre la méthode appelante et cette méthode **m()** des sous-classes de cette classe qui figurent dans l'ensemble de la méthode appelante. Aussi, on ajoute le type et les sous-types des arguments de la méthode appelée déjà dans l'ensemble de la méthode appelante aux ensembles (un pour chaque classe instanciée contenant **m()**) de la méthode **m()**. Ensuite, le type et les sous-types des valeurs de retour de **m()** déjà dans les ensembles de cette méthode sont ajoutés à l'ensemble de la méthode appelante. Enfin, chaque classe et sous-classe présente dans l'ensemble de la méthode appelante est ajoutée aux ensembles de **m()**.

L'algorithme XTA peut être formalisé de cette manière :

```
soit R = {} ensemble des méthodes atteignables.
soit S.m() = {} et S.f = {}, respectivement, ensemble de la méthode
m() et ensemble du field f.
A -> B signifie ajout de A dans B.
A ->> B signifie arête ajoutée entre A et B.
sous-types(c) désigne type(c) uni à tous les sous-types de c.

public static void main(String[]) -> R

pour chaque m() dans R :
        pour chaque new C() dans m() :
                C -> S.m()
        pour chaque x = c.f dans m() :
                pour chaque C dans S.f :
                        C -> S.m()
        pour chaque c.f = x dans m() :
                pour chaque C dans sous-types(f) :
                        si C dans S.m() :
                                C -> S.f
        pour chaque c.n() dans m() :
                pour chaque C dans sous-types(c) :
                        si C dans S.m() :
                                C.n() -> R
                                m() ->> C.n()
                                C -> S.C.n()
                                pour chaque A dans
                                sous-types(arguments(C.n()) :
                                        si A dans S.m() :
                                                A -> S.C.n()
                                pour chaque N dans
                                sous-types(retour(C.n()) :
```

```
si N dans S.C.n() :
    N -> S.m()
```

L'algorithme 0-CFA, selon le papier de Jean PRIVAT, est un algorithme qui construit un réseau d'entités typables où une arête signifie une inclusion des types d'une entité dans ceux d'une autre. L'algorithme construit également en parallèle un graphe d'appels où l'idée de base est de créer une arête d'une méthode A vers une méthode B lorsque cette méthode A appelle la méthode B. Pour les langages orientés objets, le réseau d'entités typables est construit selon trois règles (extraites de [29] mais reformulées pour le contexte de ce travail) :

- Si une variable se voit assigner le résultat d'une instanciation, on ajoute le type de la classe instanciée à cette variable.
- Si une variable se voit assigner une expression (autre qu'une instanciation), on ajoute le type de l'expression à cette variable.
- À chaque expression correspondant à l'appel d'une méthode **n()**, on ajoute une arête entre les types effectifs des arguments des méthodes **n()** (c'est-à-dire les méthodes des sous-classes du type effectif de la classe de **n()** également) et les paramètres formels correspondants de ces méthodes. On ajoute aussi une arête entre le type effectif de la valeur de retour de ces méthodes **n()** et le type de retour statique apparaissant dans la signature de ces méthodes **n()**.

Dans le contexte de ce travail, l'avantage de l'algorithme 0-CFA est qu'il permet de voir les types effectifs des arguments avec lesquels une méthode est appelée et le type effectif de la valeur de retour de cette même méthode.

Le premier défi du travail est d'adapter ces algorithmes pour les applications Android et plus précisément pour le *bytecode* Dalvik.

Les outils d'analyse statique pouvant construire des graphes d'appels des applications Android utilisés à l'heure actuelle font parfois apparaître une ou plusieurs des spécificités d'Android citées ci-dessus dans le graphe d'appels mais aucun ne les fait apparaitre toutes en même temps dans ce dernier (cf. Section 3 : problématique détaillée). C'est la raison pour laquelle il était utile de construire un outil permettant de visualiser toutes ces spécificités en même temps dans le graphe pour pouvoir découvrir des appels de méthodes qui seraient effectués implicitement mais qui n'apparaitraient pas dans un graphe d'appels en temps normal. Par exemple, à chaque fois qu'un appel à la méthode **setContentView** est effectué dans une activité, cette méthode **setContentView** appelle elle-même les constructeurs des éléments graphiques (TextView, Button, ConstraintLayout, ...) présents dans le fichier de *layout* correspondant à cette activité. Le rôle principal d'une activité est de fournir une fenêtre dans laquelle l'application affiche son interface utilisateur [2](chaque activité fournit une interface différente). Or, lorsqu'on observe les graphes d'appels construits avec des outils d'analyse statique utilisés actuellement (cf. Sous-section 2.2), ces appels n'apparaissent pas.

2. https ://developer.android.com/guide/components/activities/intro-activities

Les *intents* sont une autre caractéristique des applications Android. Ils servent à changer d'activité, à lancer un service ou à émettre un message de *broadcast* destiné à des BroadcastReceivers (composant de base Android servant à recevoir des événements systèmes ou d'applications lorsque il est enregistré pour recevoir ce type d'événements : [19]).

La présence d'un *intent* se détecte par l'occurrence de la méthode **startActivity**, **startActivityForResult**, **startService**, **bindService**, **sendBroadcast** ou **sendOrderedBroadcast**. Ces six méthodes prennent un objet Intent en argument.

Il existe deux grandes classes d'*intents* : les *intents* explicites et les *intents* implicites. Les *intents* explicites peuvent être utilisés pour changer d'activité (via la méthode **startActivity**), changer d'activité et recevoir un résultat à la fin de la nouvelle activité (via **startActivityForResult**), lancer un service (via **startService** ou **bindService**) et émettre un message de broadcast (via **sendBroadcast** ou **sendOrderedBroadcast**). Les *intents* implicites peuvent être également utilisés pour tout cela sauf pour lancer un service. La raison sera expliquée un peu plus bas. La différence entre les *intents* explicites et implicites est que pour les *intents* explicites, il faut donner la classe de l'activité, du service ou du BroadcastReceiver à lancer/activer en argument de l'*intent*. Pour les *intents* implicites, une description (en mentionnant certaines caractéristiques : cf. 4.6) de l'activité ou des BroadcastReceivers à activer est fournie à l'*intent*.

La plupart du temps, les caractéristiques d'une activité ou d'un BroadcastReceiver sont décrites, respectivement, via les Sections *activity* et *receiver* du fichier *AndroidManifest.xml* qui, selon [12], contient le nom de package de l'application, ses composants (activités, services et BroadcastReceivers), les permissions de cette activité (par exemple, l'accès à Internet) et les caractéristiques matérielles et logicielles qu'elle doit remplir.

Des exemples d'*intents* explicites et implicites seront donnés dans la Sous-section 4.6. D'après [11], les *intents* explicites sont utilisés soit avec le constructeur *Intent(String)* mais où la chaîne de caractère en paramètre du constructeur est un nom de classe, soit avec le constructeur*Intent(Context, Class)* ou, plus rarement, *Intent(String, Uri, Context, Class)*. Tandis que les *intents* implicites sont utilisés avec le constructeur *Intent(String)* également mais où la chaîne de caractère est une action (cf. 4.6) ou avec le constructeur *Intent()* ou avec *Intent(String,Uri)*. Il est dangereux d'utiliser des *intents* implicites pour lancer des services car, sans la classe complète du service en argument, si plusieurs services remplissent les mêmes caractéristiques données dans la description du service, le service à utiliser sera indéfini et ne pourra pas être lancé.

Les *intents* peuvent aussi cacher des appels entre, par exemple, la méthode **startActivity** et les méthodes **onCreate** et **onStart** (cf. Sous-section 4.6), ce qui n'apparaît pas dans le graphe des outils actuels.

Sans oublier la réflexion qui nécessite de créer une arête explicite entre le méthode **invoke** et la méthode appelée implicitement via cette méthode, ce que les outils d'analyse statique pouvant générer des graphes d'appels ne font pas tous. La réflexion est le fait de créer et/ou d'appeler une méthode ou classe dynamiquement (c'est-à-dire au moment de l'exécution). Elle n'est pas un élément spécifique aux applications Android à proprement parler mais elle peut y figurer. Il est donc important de la prendre en compte dans la construction du graphe d'appels. On repère l'appel d'une méthode via la réflexion grâce à la présence de la méthode **invoke** de la classe **java.lang.reflect.Method**. Cette dernière prend en arguments la classe de la méthode à invoquer au moment de l'exécution ainsi que les arguments de la méthode. Le nom de la méthode à invoquer est l'argument de l'appel à **getDeclaredMethod** ou **getMethod**.

L'ajout de toutes les arêtes implicites cachées augmentent la fidélité avec laquelle le graphe d'appels décrit l'application.

L'outil utilisé dans ce travail est Rundroid : un outil développé en Java capable d'extraire plusieurs caractéristiques statiques d'applications Android (voir la description plus détaillée en début de Section 4). Le travail présenté dans cet article est une extension de cet outil et a donc été développé exclusivement en Java via l'IDE Eclipse.

Les trois sources les plus utilisées dans cette partie de développement du travail sont la liste officielle des différents *opcodes* du *bytecode* Dalvik ([17]), le papier de Tip et Palsberg ([31]) et celui de Jean PRIVAT ([29]) présentant un algorithme pour 0-CFA.

Avant de commencer le développement, les trois outils actuels de constructions de graphes d'appels : Soot, WALA et AndroGuard ont été étudiés de manière à comprendre leur fonctionnement et leurs lacunes par rapport à l'outil Rundroid étendu. Ensuite, les étapes pour étendre l'outil Rundroid ont été les suivantes :

- Premièrement, le début d'algorithme RTA présent dans l'outil a été étendu de manière à produire un graphe d'appels directement visualisable après sa construction et en faisant apparaître les caractéristiques d'Android mentionnées plus haut dans le graphe. Les différentes caractéristiques d'Android ont été ajoutées dans cet ordre : les points d'entrées multiples, les fichiers de *layout*, les méthodes appelées via la réflexion et les *intents*.
- Deuxièmement, l'algorithme XTA a été rajouté dans l'outil.
- Troisièmement, l'algorithme 0-CFA a à son tour été rajouté dans Rundroid.

2 Travaux sur les graphes d'appels des applications Android

2.1 Outil de construction de graphes d'appels le plus utilisé

La revue systématique de la littérature la plus complète sur les techniques d'analyse statique des applications Android (et la plus récente) est celle de Li LI et al. [23] de 2017. On y apprend que c'est le *framework* Soot : [14] (*framework* qui analyse et transforme le *bytecode* Java à des fins d'optimisations) avec sa représentation intermédiaire (transformation intermédiaire du *bytecode* Dalvik) Jimple qui est le plus utilisé comme outil de support pour l'analyse statique d'applications Android et notamment pour la construction de graphes d'appels cette année là (en 2017).

Par le paragraphe précédent, nous pouvons constater que Soot était l'outil le plus utilisé pour l'analyse statique d'applications Android et pour générer des graphes d'appels en 2017. Mais qu'en est-il à l'heure actuelle ? Pour y répondre, des articles de recherche récents (2018 comme date limite d'ancienneté) ont été utilisés.

A l'heure actuelle, Soot et sa représentation intermédiaire Jimple restent un des outils sur lequel beaucoup de nouveaux outils s'appuient. Parmi ceux-ci, nous pouvons citer ARPDroid ([4]) : un outil pour détecter les incompatibilités entre les permissions à l'exécution des applications Android qui utilise Soot pour générer le graphe d'appels.

Nous pouvons également citer IctApiFinder ([7]) : un outil calculant les versions des systèmes d'exploitations dans lesquelles une API d'une application Android peut être invoquée et qui utilise Soot pour construire un graphe de contrôle de flux interprocédural (en rappelant qu'un graphe d'appels est un sous-graphe du graphe de contrôle de flux).

Il y a aussi Ripple ([33]) : un outil pour analyser la réflexion dans les applications Android dans des contextes où des flux d'informations explicites manquent (comme lors d'appels à des *intents*) qui utilise Soot et FlowDroid ([2]) combinés pour construire un graphe d'appels.

Parmi eux se trouve aussi ConsiDroid ([5]) : un outil qui utilise l'exécution concolique (qui mélange l'exécution abstraite et concrète) pour détecter les vulnérabilités aux injections SQL et qui utilise Soot pour générer des graphes d'appels.

Pour finir, il y a également un outil de génération automatique de comportements potiellement non autorisés mais possibles d'une application : [34] qui se base aussi sur Soot pour construire un graphe d'appels.

2.2 Outils d'analyse statique utilisés actuellement générant des graphes d'appels

Puisque l'objectif est de, au final, participer à la création d'un outil générant des graphes d'appels d'applications Android tenant compte des spécificités d'Android comme les *intents*, les points d'entrées multiples, etc, il serait intéressant d'analyser quelques outils existants.

Soot (mentionné en Sous-section 2.1) est l'outil le plus utilisé pour générer des graphes d'appels d'applications Android mais il en existe deux autres utilisés à l'heure actuelle. Ces deux autres outils sont WALA ([20]) et AndroGuard ([9]).

Soot est un outil d'analyse et d'optimisation des applications Java et, depuis quelques temps, Android. Soot permet de créer des graphes d'appels d'une application entière mais aussi des *control flow graphs* (graphes de flux de contrôle) de chaque méthode. Lorsque Soot est sollicité pour la création d'un de ces types de graphe, il met le résultat dans un fichier .dot et ce dernier peut ensuite être visualisé via un outil comme Graphviz ([18]).

Soot permet de générer des graphes d'appel via les algorithmes CHA, RTA, VTA (voir [30]), 0-CFA et k-CFA (extension de 0-CFA où les contextes dans lequels les méthodes sont appelées sont retenus : [29]). Cependant, pour pouvoir créer un graphe d'appels (voir figure 1) d'une application Android avec Soot il faut utiliser l'outil soot-infoflow-android ([15]) de FlowDroid.

En comparant les différents graphes d'appels de Soot (créés par les différents algorithmes ci-dessus), on observe qu'ils sont en réalité tous identiques. Cela nous amène à la conclusion que peu importe l'algorithme (la méthode) de construction de graphe d'appels de Soot, les noeuds et les arêtes trouvés sont identiques. La base des algorithmes doit donc être commune.

Pour manipuler le *bytecode* Dalvik facilement, Soot le transforme en une représen- tation intermédiaire : soit en Jimple (la plus utilisée), en Baf (représentation allégée facile à manipuler), en Grimp (version condensée de Jimple efficace pour la décompilation et l'inspection de code) ou en Shimple (variante *single static assignment* de Jimple). Soot est implémenté en Java.

WALA est un outil en Java d'analyse statique de *bytecode* Java, Javascript et désormais Android. Il permet aussi de générer des graphes d'appels (voir figure 2). Il met également le résultat dans un fichier .dot (donc à visualiser avec un outil comme Graphviz). Les types d'algorithmes de graphes d'appels qu'il peut manipuler sont CHA, RTA, 0-CFA et k-CFA.

Pour manipuler le *bytecode* Dalvik, WALA a lui aussi sa propre représentation intermédiaire : WALA IR, qui assez proche du *bytecode* JVM (en version *single static assignment*).

WALA est utilisé dans des outils récents comme SIERRA ([21]) : un outil de détection statique de situations de compétition (*races*) qui utilise WALA pour construire des graphes d'appels d'applications Android.

WALA est également utilisé dans l'algorithme présenté dans [27] qui détecte les attaques dites de "canal caché" et les fuites de données dans les applications Android et qui se sert de WALA pour construire un graphe d'appels.

Comme outil récent utilisant WALA, nous pouvons également citer PIKADROID ([1]) : un outil permettant de détecter les *malwares* dans des API appelées par des applications et qui utilise notamment un graphe d'appels construit par WALA pour cela.

Il y a aussi l'algorithme de *clustering* des librairies tierces présenté dans [24] qui utilise WALA pour la construction d'un graphe d'appels des API.

Enfin, nous pouvons mentionner ADLIB ([22]) : un outil analysant les librairies Java d'Android des plateformes publicitaires pour détecter des flux éventuels d'API tierces vers des fonctionnalités comme la location géographique et qui se sert de WALA pour construire les graphes d'appels du SDK de la plateforme publicitaire.

AndroGuard, quant à lui, est un outil en Python permettant d'analyser exclusivement les applications Android. Il accepte le format dex, odex, apk et xml.

On dénombre cinq grandes fonctionnalités pour cet outil :

- l'identification des classes appelées depuis une autre classe et les endroits à partir desquels des String sont utilisées
- le parsing du *bytecode*
- la décompilation et le désassemblage
- l'affichage des informations sur les certificats des applications Android et le décodage (parsing) du xml
- la construction de *control flow graphs* d'instructions niveau assembleur
- la construction de graphes d'appels (voir figure 3)

AndroGuard crée les graphes d'appels notamment sous le format gml, gexf ou graphml. D'autres formats existent mais ce sont les formats les plus utilisés car ils peuvent être lus par l'outil Gephi ([13]).

AndroGuard ne possède pas de représentation intermédiaire, il manipule le *bytecode* Dalvik directement.

AndroGuard est utilisé par des outils récents comme DroidEnsemble ([32]) : un outil analysant les chaînes de caractères et les caractéristiques structurelles pour détecter de manière précise les *malwares* dans les applications et qui utilise AndroGuard pour extraire le graphe d'appels des applications.

AndroGuard est également utilisé par AndrEnsemble ([26]) : un système de caractérisation des familles de *malwares* basé sur les ensembles d'appels à des API suspectes extraits via des graphes d'appels de différentes familles aggrégés. Ces graphes d'appels étant construits avec AndroGuard.

AndroGuard est aussi utilisé dans l'algorithme présenté dans [25] permettant de détecter les *malwares* en combinant des techniques d'analyse statique et dynamique d'applications qui utilise AndroGuard pour construire le graphe d'appels de ces applications.

Nous pouvons également citer PlumbDroid ([3]) : un outil qui répare les fuites de ressources de manière automatisée dans les applications Android et qui utile AndroGuard pour construire le graphe de contrôle de flux des méthodes.

Enfin, l'algorithme présenté dans [28] utilise également AndroGuard. Cet algorithme consiste à analyser les structures ayant des noeuds fortement connectés et qui ont des similarités entre eux dans les graphes d'appels afin de détecter des *malwares* dans des applications Android. Ces graphes d'appels sont construits avec AndroGuard.

En ce qui concerne les algorithmes de graphes d'appels, l'outil le plus complet est Soot car c'est celui qui possède le plus large panel d'algorithmes (cinq algorithmes), comparé à WALA (quatre algorithmes) et AndroGuard (un seul algorithme !).

Il faut noter que AndroGuard considère toutes les librairies Java et Android ainsi que les appels entre elles. Ce qui donne un graphe d'appels très volumineux.

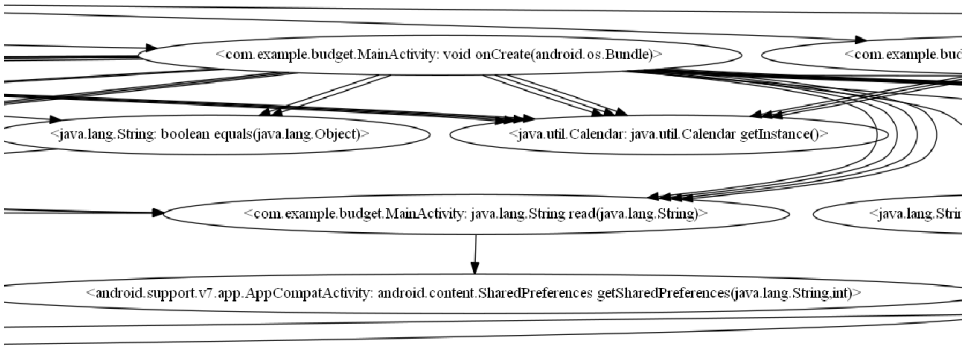

FIGURE 1 – zoom dans le graphe d'appels de Soot

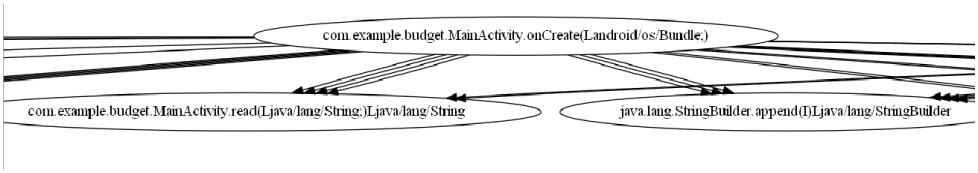

FIGURE 2 – zoom dans le graphe d'appels de WALA

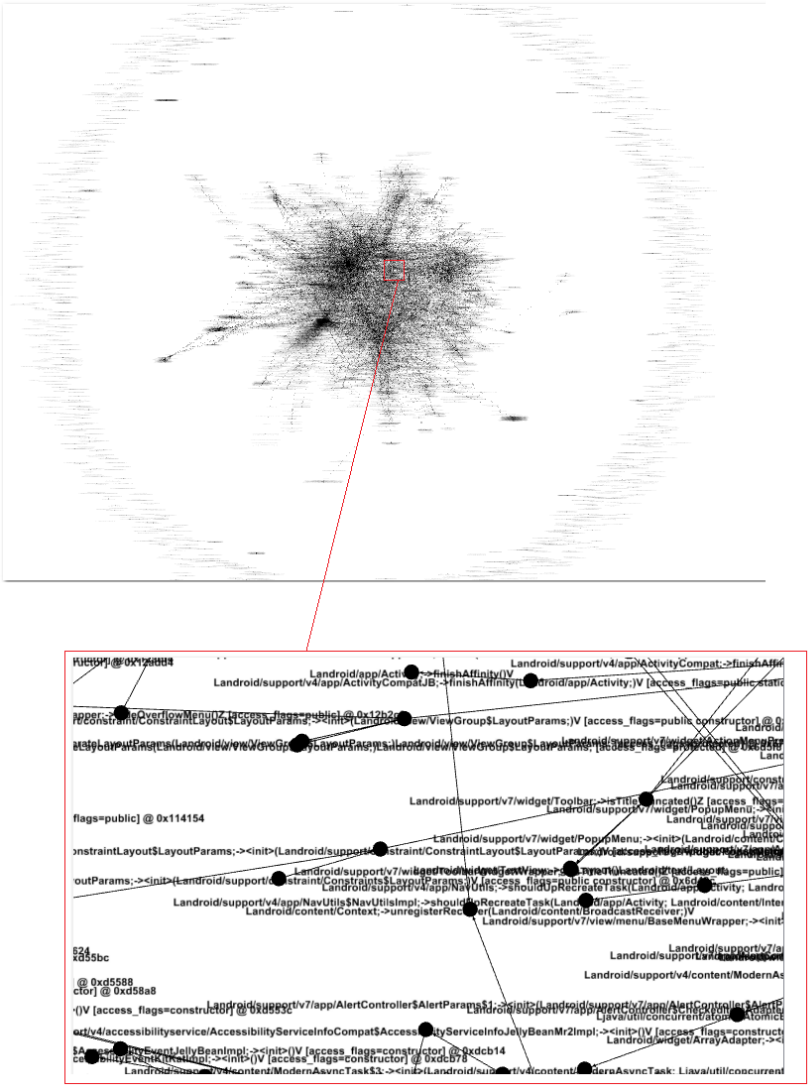

FIGURE 3 – graphe d'appels de AndroGuard

Nous avons pu voir, dans cette section, que la construction des graphes d'appels est beaucoup utilisée dans le cadre de la détection de *malwares* dans les

applications Android. Ces graphes sont aussi utlisés pour détecter les fuites de données sensibles, les injections SQL, les situations de compétition et encore bien d'autres problèmes. Cela explique en quoi la construction de graphes d'appels est utile et pourquoi un graphe détaillé pourrait apporter beaucoup aux outils décrits dans cette Section.

3 Problématique détaillée

Les outils Soot, WALA et AndroGuard, décrits dans la Section précédente, ne tiennent pas compte de toutes les spécificités des applications Android en construisant leurs graphes d'appels.

Le tableau 1 établit une comparaison entre les trois outils et les spécificités d'Android qu'ils incluent chacun dans leur graphe d'appels.

Elements Android	Soot	WALA	AndroGuard
Points d'entrée multiples	oui	oui	oui
Ressources xml	non	non	non
Reflexion	non (mais log des sites d'appels reflexifs)	oui (mais à activer)	non
Intents	non	non	non
Bibliothèque Android	oui	oui	oui

TABLE 1 – Comparaison entre les graphes d'appels de Soot, WALA et AndroGuard

Soot, WALA et AndroGuard gèrent tous les trois les points d'entrée multiples. Les trois outils parsent le fichier *AndroidManifest.xml* pour récupérer les points d'entrée.

Pour Soot, voici le code de récupération des points d'entrée (dans la classe *soot.jimple.infoflow.android.SetupApplication* de l'outil soot-infoflow-android) :

```
...
this.manifest = new ProcessManifest(apkFileLocation);
this.entrypoints = new HashSet<>();
for (String className : manifest.getEntryPointClasses())
        this.entrypoints.add(Scene.v().getSootClassUnsafe(className));
...
```

Pour WALA, le code de récupération des points d'entrée se trouve dans la classe *org.scandroid.util.EntryPoints* :

```
...
for (String[] intent : ActivityIntentList){
      // method = IntentToMethod(intent[0]);
      method = "onCreate(Landroid/os/Bundle;)V";

      im = cha.resolveMethod(StringStuff.makeMethodReference(intent[1
      + '.'+
```

```
        method));
        if (im!= null)entries.add(new DefaultEntrypoint(im, cha));
}
...
```

Concernant AndroGuard, l'extraction des points d'entrées s'effectue dans le fichier *'python3.x'(dossier où python est installé)/lib/site-packages/AndroGuard/cli/main.py* :

```
...
a, d, dx = AnalyzeAPK(APK)

entry_points = map(FormatClassToJava,
                           a.get_activities()+ a.get_providers()+
                           a.get_services()+ a.get_receivers())
entry_points = list(entry_points)

log.info("Found The following entry points by search AndroidManifest.xml : "
             "{}".format(entry_points))
...
```

Aucun des trois outils n'inclut les informations de *layout* des ressources xml dans le graphe d'appels.

AndroGuard permet de décompiler un fichier .apk et d'ensuite avoir accès aux ressources xml mais n'inclut en aucun cas des informations de celles-ci dans son graphe d'appels.

Pour ce qui est de la reflexion, Soot n'inclut pas les appels aux méthodes appelées par ce mécanisme dans le graphe d'appels mais permet de logger les appels reflexifs.

WALA permet d'inclure les appels reflexifs dans le graphe d'appel mais il faut le préciser dans le code via ces deux lignes :

```
AnalysisOptions options = new AnalysisOptions(this.scope,
this.entrypoints);
options.setReflectionOptions(AnalysisOptions.ReflectionOptions.FULL);
```

AndroGuard, de son côté, n'inclut pas les appels reflexifs dans le graphe d'appels.

Concernant les *intents*, ni Soot, ni WALA, ni AndroGuard ne les inclut dans son graphe d'appels.

Enfin, les trois outils tiennent compte des librairies Android dans le graphe d'appel. Soot et WALA permettent même, avec de simples conditions sur les noms des noeuds générés, de filtrer les librairies qu'on souhaite inclure dans le graphe d'appels.

AndroGuard, lui, ne permet pas directement de filtrer les librairies utilisées lors de la construction du graphe d'appel. Il inclut toutes les librairies et même les appels entre elles, ce qui produit, comme mentionné dans la Section précédente, un graphe d'appels très conséquent.

Ces observations nous mènent à la conclusion qu'il manque un outil qui serait capable de prendre en compte toutes les spécificités d'Android dans les graphes d'appels qu'il génère.

4 Amélioration de l'outil Rundroid

En ayant pris conscience de la problématique, le but est maintenant de créer un outil capable de créer des graphes d'appels qui tiennent compte de toutes les spécificités d'Android.

Le travail a été de partir d'un outil existant : Rundroid qui permet déjà, à partir d'un fichier .apk :

- d'afficher les noms des fichiers de *layout*
- d'afficher chaque classe (fichier .java) du fichier et son *bytecode* Dalvik en parsant le fichier *classes.dex* de l'apk
- d'afficher les classes de l'apk qui ne sont pas des librairies
- de faire de l'exécution symbolique du *bytecode* en traversant les différentes classes de l'apk
- de transformer le *bytecode* Dalvik en programme logique avec contraintes

Les différentes fonctionnalités citées au paragraphe précédent et surtout le parsing du *bytecode* Dalvik à l'état binaire (pour ensuite l'afficher de manière lisible), indiquent que la construction d'un graphe d'appels de l'apk est possible (en analysant notamment les différents appels à *invoke-virtual*, *invoke-super*, *invoke-direct*, *invoke-static* et *invoke-interface* (voir 4.1) et l'analyse de l'assignation en amont de leur paramètres pour obtenir les différents appels de méthodes dans l'apk).

Une implémentation de l'algorithme RTA conforme à l'algorithme présenté dans le papier de Tip et Palsberg ([31]) est présente dans Rundroid mais ne prend pas en compte les spécificités d'Android mentionnées dans la Section précédente.

Rundroid permet également de parser les fichiers xml via le parseur xml de android4me ([10]). Cela sera utile pour analyser le contenu des fichiers du dossier res/layout (dossier contenant les fichiers de *layout*) et le prendre en compte dans le graphe d'appels. La procédure exacte sera décrite ultérieurement.

4.1 Élements essentiels du *bytecode* Dalvik

Le *bytecode* Dalvik est le *bytecode* dans lequel les applications Android sont écrites (cf. [17]). Il est crucial de le comprendre car ce n'est pas à partir du code de haut niveau en Java de l'application que le graphe d'appels sera construit mais bien à partir de ce *bytecode* Dalvik.

Le *bytecode* Dalvik est basé sur les registres ([17]), la plupart des instructions sont donc composées d'un ou plusieurs registres sources (contenant une opérande) et d'un registre cible. Chaque registre du programme a un nom composé

d'un "v" et d'un nombre correspondant au numéro du registres. Les arguments sont des registres spéciaux commençant par la lettre "p" au lieu de "v". Dans la suite de ce travail, ces registres seront appelés variables.

Si nous reprenons l'algorithme RTA, les instructions les plus importantes sont l'instruction d'instanciation de classe et les instructions d'appel à une méthode.

L'instruction d'instanciation est appelée "new-instance". L'instruction "new-instance" est composée d'un registre cible (qui accueillera la nouvelle instance) et d'une opérande (qui n'est pas un registre) qui est le nom de la classe (ou le type) à instancier. Sa structure est la suivante :

new-instance vAA, type

où "A" représente un chiffre et où "type" est le type à instancier. Par exemple, le code Java :

```
A a = new A();
```

sera traduit en *bytecode* Dalvik de la manière suivante :

```
new-instance v1, Lcom/app/A;
```

À noter que la variable n'est pas toujours "v1", c'est le nom qui lui a été attribué pour l'exemple. À chaque fois que cette instruction est rencontrée dans une méthode atteignable dans l'algorithme RTA, l'opérande de l'instruction (la classe à instancier) est ajoutée à l'ensemble des classes instanciées. Cette fonctionnalité était déjà disponible dans la version d'origine (avant de commencer le travail) de Rundroid.

Les instructions d'appel à une méthode sont "invoke-super", "invoke-direct", "invoke-virtual", "invoke-static" et "invoke-interface".

L'instruction "invoke-super" correspond à l'appel d'une méthode du super-type.

L'instruction "invoke-direct", elle, correspond à l'appel d'une méthode privée ou d'un constructeur. Cette méthode doit avoir des paramètres ([17]).

L'instruction "invoke-virtual" correspond à l'appel d'une méthode ni statique, ni privée ni *final* avec des paramètres ([17]).

L'instruction "invoke-static" correspond à l'appel d'une méthode statique ([8]).

L'instruction "invoke-interface", quant à elle, correspond à l'appel d'une méthode d'une interface ([8]).

Toutes les instructions d'appels de méthode ont la même structure :

invoke-kind {arguments}, method_name

où "kind" est soit "super", soit "direct", soit "virtual", soit "static" ou soit "interface" , "arguments" sont les registres passés en arguments de la méthode à appeler et "method_name" est le nom de la méthode à appeler. Dans le *bytecode* Dalvik, le nom des méthodes doit toujours comprendre le package et la classe de la méthode également et être écrit selon la convention

L*nom-de-package1*/*nom-de-package2*/.../*nom-de-la-classe*;->abstract final *public* | *private* | *protected nom-de-la-methode*(*type-paramètre1 type-paramètre2...*)*type-de-retour*.

Concernant les types des paramètres et le type de retour, ils peuvent être des types primitifs ou des types objets. Si il s'agit de types primitifs, ils seront représentés par une lettre [3] : V (*void* : uniquement pour les types de retour), Z (booléen), B (*byte*), S (*short*), C (*char*), I (*int*), J (*long*), F (*float*) ou D (*double*). Si il s'agit de types objets, ils s'écrivent selon la convention :

L*nom-de-package1*/*nom-de-package2*/.../*nom-de-la-classe*;

Par exemple, le code Java :

```
a.m();
```

sera traduit en *bytecode* Dalvik de la manière suivante :

```
invoke-virtual {v1} Lcom/app/A;->m()V
```

où "v1" représente, dans cet exemple, la variable d'instance "a".

Dans l'algorithme RTA, à chaque fois qu'une des instructions d'appel de méthode est rencontrée dans une méthode atteignable, on vérifie si la classe de la méthode appelée (l'opérande de l'instruction) fait partie de l'ensemble des classes instanciées. Si c'est le cas, une arête est ajoutée entre la méthode atteignable et la méthode appelée et cette dernière est ajoutée à l'ensemble des méthodes atteignables (si elle n'y figure pas). Une arête est également ajoutée entre la méthode appelante et la méthode **m()** ayant la même signature que la méthode appelée et dont la classe est C (où C est une sous-classe de la classe de la méthode appelée qui fait partie de l'ensemble des classes instanciées). **m()** est aussi ajoutée à l'ensemble des méthodes atteignables si elle n'y figure pas. Cette fonctionnalité était également déjà présente dans la version d'origine de Rundroid. Cependant, dans le contexte de ce travail, pour alléger le graphe d'appels, **m()** ne sera prise en compte que si elle est déclarée explicitement dans le code et qu'elle n'existe pas juste implicitement par une relation d'héritage (cf. 4.3).

Concernant l'algorithme XTA, les instructions les plus importantes sont "new-instance", "invoke-kind" (où kind peut être *super*, *direct*, *virtual*, *static* ou *interface*), les instructions de lecture dans un champ et les instructions d'écriture dans un champ.

3. https ://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields

Les instructions de lecture dans un champ sont "iget" (pour les champs entiers), "iget-short" (pour les champs entiers de 16 bits en complément à 2), "iget-wide" (pour les champs de type *long* ou *double*), "iget-object" (pour les champs qui ont un type objet), "iget-boolean" (pour les champs booléens), "iget-byte" (pour les champs de type *byte*) et "iget-char" (pour les champs de type *char*). Si le champ est statique, les instructions "sget" (pour les champs entiers), "sget-short", "sget-wide", "sget-object", "sget-boolean", "sget-byte" et "sget-char" sont utilisées. Ces instructions permettent de récupérer la valeur d'un champ et de la mettre dans une variable).

L'instruction *iget* et ses dérivés ont toutes la même structure :

iget-kind vAA, vBB, field_name

où "A" et "B" sont des chiffres ("AA" signifie qu'on peut mettre un nombre de deux chiffres maximum comme numéro de registre), "kind" est "short", "wide", "object", "boolean", "byte" ou "char", "vAA" est la variable cible, "vBB" est la variable contenant l'instance de la classe du champ et "field_name" est le champ dont on veut extraire la valeur. Un champ est toujours noté selon la convention suivante :

L/*package1*/*package2*/.../*classe_du_field*->*nom_du_field* :*type_du_field*

Par exemple, le code Java :

```
b = a.i;
```

où "i" est un entier, se traduit en *bytecode* Dalvik de la manière suivante :

```
iget v2, v1, Lcom/app/A;->i :Lcom/app/A;
```

où "v2" est la variable "b" et "v1" est la variable "a".

L'instruction "sget" et ses dérivés ont également toutes la même structure :

sget-kind vAA, field_name

où "A" est un chiffre, "kind" est "short", "wide", "object", "boolean", "byte" ou "char", "vAA" est la variable cible, et "field_name" est le champ dont on veut extraire la valeur. Par exemple, le code Java :

```
b = A.i;
```

se traduit en *bytecode* Dalvik de la manière suivante :

```
sget v1, Lcom/app/A;->i :Lcom/app/A;
```

où "v1" est la variable "b".

Les instruction d'écriture dans un champ sont "iput" (pour les champs entiers), "iput-short" (pour les champs entiers de 16 bits en complément à 2), "iput-wide" (pour les champs de type *long* ou *double*, "iput-object" (pour les champs qui ont un type objet), "iput-boolean" (pour les champs booléens), "iput-byte" (pour les champs de type *byte* et "iput-char" (pour les champs de

type *char*). Si le champ est statique, les instructions "sput" (pour les champs entiers), "sput-short", "sput-wide", "sput-object", "sput-boolean", "sput-byte" et "sput-char" sont utilisées. Ces instructions permettent d'assigner la valeur d'une variable à un champ.

La structure de *iput* et ses dérivés est la suivante :

iput-kind vAA, vBB, field_name

où "A" et "B" sont des chiffres, "kind" est "short", "wide", "object", "boolean", "byte" ou "char","vAA" est la variable dont la valeur sera assignée au champ, "vBB" est la variable contenant l'instance de la classe du champ et "field_name" est la champ dans lequel on veut inscrire une valeur. Par exemple, le code Java :

```
a.i = b;
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
iput v2, v1, Lcom/app/A;->i :Lcom/app/A;
```

où "v2" est la variable "b" et "v1" la variable "a".

La structure de *sput* est ses dérivés est la suivante :

sput-kind vAA, field_name

où "A" est un chiffre, "vAA" est la variable dont la valeur sera assignée au champ et "field_name" est le champ dans lequel la valeur sera inscrite. Par exemple, le code Java :

```
A.i = b;
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
sput v1, Lcom/app/A;->i :Lcom/app/A;
```

où "v1" est la variable "b".

Pour l'algorithme 0-CFA, les trois instructions essentielles sont "new-instance", "invoke-kind" (où kind peut être *super*, *direct*, *virtual*, *static* ou *interface*) et les instructions d'assignation ("new-instance", "iget" et ses dérivés et "sget" et ses dérivés font d'ailleurs partie de ce type d'instruction).

Les instructions d'assignation d'objets (les types primitifs ne sont pas pris en compte dans le contexte de ce travail) les plus courantes (excepté "new-instance", "iget-object" et "sget-object") sont "move-object", "move-object/from16", "move-result-object", "const-string", "const-class", "new-array" et "aget-object".

La raison pour laquelle les types primitifs ne sont pas pris en compte dans le graphe est que, dans notre contexte, il est plus intéressant de connaître les types objets effectifs plutôt que les types primitifs effectifs des arguments d'une méthode. En effet, si le type d'un paramètre formel d'une méthode est *int*, nous pouvons être certains que l'argument de cette dernière sera de type *int*. Par contre, si le type de ce même paramètre formel est *java.lang.Object*, l'argument

pourrait être de n'importe quel sous-type de *Object*. Il est donc intéressant, dans ce cas, de connaître le type effectif de l'argument.

Les instructions "move-object/from16" et "move-object" permettent toutes les deux de mettre la valeur d'une variable assignée à un objet dans une autre variable. La différence entre les deux est que le registre de destination de "move-object/from16" peut être composé de 8 bits et son registre source peut être composé de jusqu'à 16 bits. Ce n'est pas le cas pour "move-object" où les registres source et destination sont limités à 4 bits ([17]).

La structure des deux instructions citées dans le paragraphe précédent est la suivante :

move-object(/from16) vA(A), vB(BBB)

où "A" et "B" sont des chiffres, "vA(A)" est le registre destination et "vB(BBB)" est le registre source (à transférer dans le registre destination). Par exemple, le code Java :

```
A a = new A();
B b = new B();
a = b;
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
new-instance v1, Lcom/app/A;
new-instance v2, Lcom/app/B;
move-object v1, v2
```

L'instruction "move-result-object" est une instruction utilisée après l'appel à une méthode qui retourne un objet ou une instruction de création d'un tableau prérempli ("filled-new-array" ou "filled-new-array-range"). Son rôle est de transférer le résultat d'un appel à une méthode retournant un objet ou un tableau prérempli dans une variable.

La structure de "move-result-object" est la suivante :

move-result-object vA

où "A" est un chiffre et "vA" est la variable cible (dans laquelle mettre le résultat de l'appel ou le tableau prérempli). Par exemple, le code Java :

```
b = a.m2();
```

où "b" est de type B, "a" est de type A est la valeur de retour de **m2()**, est de type B est traduit en *bytecode* Dalvik de la manière suivante :

```
invoke-virtual {v1}, Lcom/app/A;->m2()Lcom/app/B;
move-result-object v2
```

où "v1" correspond à la variable "a" et "v2" correspond à la variable "b".

L'instruction "const-string" est utilisée pour assigner une chaîne de caractères à une variable. Sa structure est la suivante :

const-string vA, string

où "A" est un chiffre, "vA" est la variable cible et "string" est la chaîne de caractères à assigner à cette variable. Par exemple, le code Java :

```
String s = "bonjour";
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
const-string v3, "bonjour"
```

où "v3" correspond à la variable "s".

L'instruction "const-class" permet d'assigner une classe à une variable. Sa structure est la suivante :

const-class vA, class

où "A" est un chiffre, "vA" est la variable cible et "class" est la classe à assigner à cette variable. Par exemple, le code Java :

```
Class c = A.class;
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
const-class v4, Lcom/app/A;
```

où "v4" correspond à la variable "c".

L'instruction "new-array" est une instruction servant à créer un tableau vide d'un type et d'une taille donnés. Sa structure est la suivante :

new-array vA, vB, type

où "A" et "B" sont des chiffres, "vA" est la variable cible, "vB" est la variable contenant la taille du tableau et "type" est le type des éléments tableau précédé de "[". Par exemple, le code Java :

```
String[] s = new String[2];
```

est traduit en *bytecode* Dalvik de la manière suivante :

```
const/4 v2, 0x02
new-array v3, v2, [Ljava/lang/String;
```

où "const/4" est une instruction qui assigne une valeur entière à une variable et "v3" correspond à la variable "s".

Enfin, l'instruction "aget-object" est une instruction servant à récupérer un élément d'un tableau et d'assigner la valeur de cet élément à une variable. Sa structure est la suivante :

aget-object vA, vB, vC

où "A", "B" et "C" sont des chiffres, "vA" est la variable cible, "vB" est la variable contenant le tableau dont il faut extraire la valeur et "vC" est l'index du tableau dont la valeur correspond à la valeur à extraire. Par exemple, le code Java :

```
String s2 = s[1];
```

où "s" est un tableau de chaîne de caractères, est traduit en *bytecode* Dalvik de la manière suivante :

```
const/4 v2, 0x01
aget-object v5, v3, v2
```

où "v5" correspond à la variable "s2" et "v3" correspond à la variable "s".

4.2 Ajout des points d'entrées multiples

L'ajout des points d'entrées multiples fut relativement aisé car l'outil Rundroid possèdait déjà une méthode permettant d'extraire les points d'entrées d'une application. Pour cela, une recherche de toutes les déclarations de méthode de l'application qui sont dans une sous-classe d'une des classes suivantes (qui sont des activités très souvent lancées en premier lieu dans une application) :

- **android.app.Activity**
- **android.accounts.AccountAuthenticatorActivity**
- **android.app.ActivityGroup**
- **android.app.AliasActivity**
- **android.app.ExpandableListActivity**
- **android.support.v4.app.FragmentActivity**
- **android.app.ListActivity**
- **android.app.NativeActivity**
- **android.app.LauncherActivity**
- **android.preference.PreferenceActivity**
- **android.app.TabActivity**
- **com.google.android.maps.MapActivity**

est effectuée puis chaque signature de méthode est inspectée pour voir si elle n'est pas égale à **protected onCreate(Landroid/os/ Bundle ;)V** ou **public onCreate (Landroid/os/Bundle ;)V** (ces méthodes sont lancées à la création d'une application). Si c'est le cas, on considère cette méthode comme un point d'entrée.

Il suffisait maintenant, pour être complet, d'ajouter les autres méthodes du cycle de vie d'une activité :

- **onStart()** (lancée quand une application est démarrée)
- **onRestart()** (lancée quand une application est redémarrée)
- **onResume()** (lancée quand une application est relancée après une pause)
- **onPause()** (lancée quand une application est mise en pause)
- **onStop()** (lancée quand une application est stoppée)
- **onDestroy()** (lancée quand une application est terminée et est détruite)

aux points d'entrées.

4.3 Ajout de la visualisation du graphe d'appels

L'implémentation actuelle fournie dans Rundroid affiche le graphe d'appels mais de manière textuelle (voir figure 4).

```
Failed: cannot find method with name_idx=61 and proto_idx=4 from class_idx=3
S = { }
I consists of:
(invoke-super,
 caller = ENCODED_METHOD{offset=1628,#bytes=4}(method_idx_diff=UNSIGNED_LEB128[1 byte(s)](value=2),method_idx=9,
 access_flags=UNSIGNED_LEB128[1 byte(s)](value=1),access_flags_name=public,code_off=UNSIGNED_LEB128[2 byte(s)](value=1856))
 static target = METHOD_ID_ITEM{offset=836,#bytes=8}(class_idx=8,proto_idx=3,name_idx=65)
 runtime targets = { })
(invoke-virtual,
 caller = ENCODED_METHOD{offset=1628,#bytes=4}(method_idx_diff=UNSIGNED_LEB128[1 byte(s)](value=2),method_idx=9,
 access_flags=UNSIGNED_LEB128[1 byte(s)](value=1),access_flags_name=public,code_off=UNSIGNED_LEB128[2 byte(s)](value=1856))
 static target = METHOD_ID_ITEM{offset=812,#bytes=8}(class_idx=8,proto_idx=1,name_idx=53)
 runtime targets = { })
```

FIGURE 4 – Exemple de graphe d'appels RTA textuel de Rundroid

Nous constatons ici que Rundroid affiche une arête sous la forme d'un tuple de la structure suivante :

(c, m(), n() où mode_appel(n()) = statique, {n'() | mode_appel(n'()) = exécution})

où "c" est le type d'appel (exemples : *invoke-super*, *invoke-virtual*, ...), "m()" est la méthode appelante, "n()" est la méthode appelée statiquement [4] par "m()", "mode_appel" est une fonction qui prend en argument une méthode appelée et renvoie soit "statique" (si la méthode est appelée statiquement) soit "exécution" (si la méthode est appelée à l'exécution) et "{n'() | mode_appel(n'()) = exécution}" sont les méthodes appelées à l'exécution par "m()".

L'étape suivante a été de rendre les informations de la figure 4 plus facilement lisibles en transformant chaque expression après *"caller ="*, *"static target ="* et *"runtime targets ="* en une expression de la forme :

"classe de la méthode"->"signature de la méthode".

Il manquait un détail à l'implémentation de l'algorithme RTA pour qu'il soit identique à celui du papier de Tip et Palsberg : il fallait prendre en compte que même si une méthode est appelée avec un type déclaré A, si une sous-classe de A possède une méthode explicitement écrite avec la même signature, une arête sera ajoutée dans le graphe entre la méthode appelante et la méthode de cette sous-classe. Pour illustrer cela, voici un exemple : si nous avons le bout de code Java suivant :

```
class Main {
        A a = new B();
        a.m();
}

class A {
        public A(){...}
        void m(){...}
}
```

4. par appel statique, il faut comprendre un appel qu'on peut déduire lors d'une analyse statique et pas obligatoirement à l'exécution

```
class B extends A{
        public B(){...}
        void m(){...}
}
```

Ce ne sera pas la méthode **m()** de la classe A qui sera prise en compte dans le graphe mais celle de la classe B (car A ne fait pas partie de l'ensemble des classes instanciées et si A était dans cet ensemble, une la méthode **m()** de A figurerait aussi dans le graphe).

L'implémentation la partie manquante de l'algorithme RTA mentionnée au paragraphe précédent se fait de la manière suivante :

1. On vérifie, pour chaque méthode appelante, si il existe des sous-classes de sa cible statique (c'est-à-dire la méthode qu'elle appelle statiquement, si elle existe) qui sont dans l'ensemble des classes instanciées.
2. Si c'est le cas, on vérifie si une méthode avec la même signature est déclarée explicitement dans les sous-classes (ici, le fait d'hériter d'une méthode des super-classes n'est pas suffisant, il faut que la méthode figure explicitement dans la classe). Le cas échéant, une flèche sera tracée de la méthode appelante vers la méthode de la sous-classe de la méthode appelée statiquement.

Le raisonnement est le même pour les méthodes appelées à l'exécution par la méthode appelante.

Remarque : dans l'algorithme RTA classique, si une méthode appelle une autre méthode m d'une classe A, les méthodes **m** (même implicites) des sous-classes de A sont aussi ajoutées au graphe.

Dans le contexte de ce travail, le choix a été fait de ne pas surcharger le graphe d'appels et les méthodes implicites des sous-classes ne figurent donc pas dans le graphe. Ce choix est surtout motivé par le fait que certaines méthodes des sous-classes n'existent pas explicitement dans le programme et ne sont souvent jamais appelées. C'est donc de l'information qu'il n'est pas nécessaire de rajouter dans le graphe d'appels.

La formalisation de l'algorithme RTA pour le *bytecode* Dalvik, dans notre contexte, est la suivante :

```
soit R = {} et S ={}, respectivement, ensemble des méthodes
atteignables et ensemble des classes instanciées.
A -> B signifie ajout de A dans B.
A ->> B signifie arête ajoutée entre A et B.
"kind" = "super", "direct", "virtual", "static" ou "interface".

public static main([Ljava/lang/String;)V -> R

pour chaque Lx/y/D;->m() dans R :
        pour chaque new-instance vA, La/b/C; dans Lx/y/D;->m() :
                La/b/C; -> S
        pour chaque invoke-kind {vA, vB, ...} Lw/z/C;->n()
        dans Lx/y/D;->m() :
                si Lw/z/C; dans S :
```

```
        Lw/z/C;->n() -> R
        Lx/y/D;->m() ->> Lw/z/C;->n()
pour chaque sous-type(Lw/z/C;) dans S :
        si n() dans sous-type(Lw/z/C;) :
                sous-type(Lw/z/C;)->n() -> R
                Lx/y/D;->m() ->>
                sous-type(Lw/z/C;)->n()
```

En connaissant toutes les méthodes appelantes et les cibles statiques et à l'exécution de ces méthodes (et en ayant rendues ces dernières lisibles), il est possible de construire la visualisation du graphe d'appels. L'approche a été de considérer chaque méthode appelante et leurs méthodes cibles statiques et à l'exécution. Ces dernières sont écrites dans un fichier .dot de la manière suivante : la méthode appelante est considérée comme l'origine d'une arête du fichier .dot et chaque méthode cible est considérée comme la destination d'une arête ayant comme origine la méthode appelante. Ce qui donne, lors de la visualisation, une flêche allant de la méthode appelante vers la méthode cible. La taille du graphe pouvant s'accroître rapidement, les noeuds racines du graphe (comprenant les points d'entrées et la méthode **onClick**) sont colorés en jaune pour plus de visibilité.

Toujours dans une optique d'augmenter la lisibilité, si un noeud source (une méthode appelante) a plusieurs fois le même noeud destination (la même méthode cible), une seule arête est tracée entre la source et la destination.

Une distinction est faite entre les méthodes appelées statiquement et celles appelées au *runtime*. Pour ce faire, les flêches partant d'une méthode appelant une méthode au *runtime* sont colorées en rouge et les flêches partant d'une méthode appelant une méthode statiquement sont colorées en noir.

La dernière étape a été d'afficher la visualisation du graphe d'appels à l'écran juste après le calcul de ce dernier. Avant de pouvoir afficher le graphe d'appels il faut convertir le fichier .dot en une image (en l'occurrence, un fichier .png ou fichier .svg). Cette conversion est faite via la commande **dot** de l'outil Graphviz. Une fois la conversion effectuée, si le fichier .dot n'est pas trop volumineux (moins de 1000 lignes) le fichier .png est affiché dans un JFrame. Par contre, si le fichier .dot dépasse 1000 lignes une conversion en un fichier .svg est effectuée et le fichier s'ouvre via le programme par défaut choisi pour lire les .svg.

Voici, ci-dessous, un exemple de graphe d'appels généré avec l'outil Rundroid étendu :

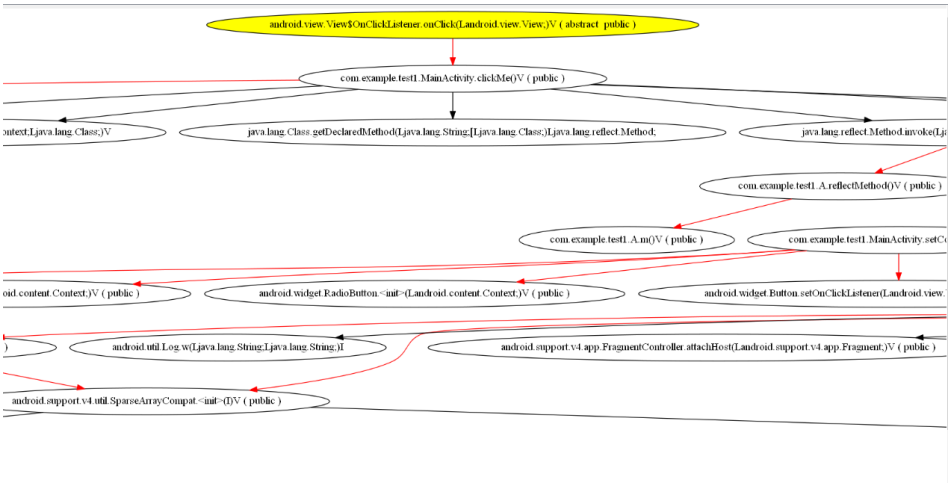

FIGURE 5 – Zoom dans le graphe d'appels de Rundroid

Le graphe d'appels suivant (figure 6), quant à lui, montre un exemple de graphe construit avec Rundroid avec uniquement les points d'entrée comme caractéristique d'Android (nœuds en jaune). Le programme utilisé pour construire ce graphe est le programme Test (cf. Section 7).

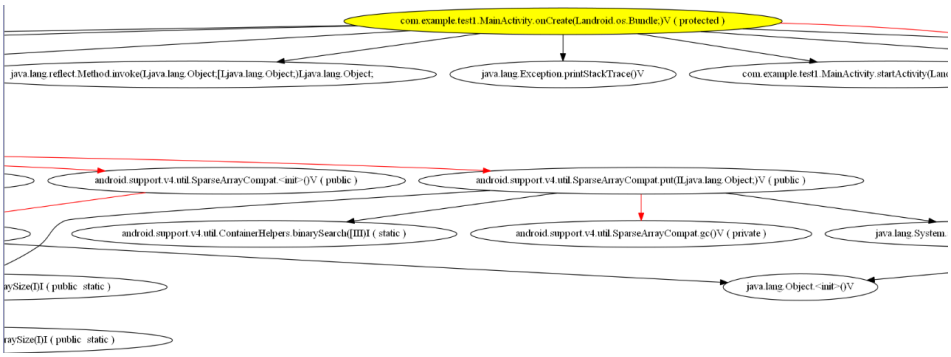

FIGURE 6 – Graphe d'appels avec les points d'entrée

4.4 Ajout des éléments de layout

Un autre élément caractéristique des applications Android est la présence des fichiers de *layout* ; fichiers xml contenant les éléments de l'interface graphique de l'application (les élément graphiques que l'utilisateur verra à l'écran) ainsi que leur disposition. Chaque activité d'une application Android possède un fichier .xml (de *layout*) qui lui est associé. Les fichiers de *layout* liés aux activités sont stockés dans le dossier *res/layout* de l'application.

Comme mentionné dans la Section précédente, les logiciels de création de graphes d'appels d'applications Android ne prennent pas en compte ces fichiers xml. Il fallait trouver un moyen de tirer pleinement profit de ces fichiers afin d'enrichir le graphe d'appels.

L'approche a été de parser les fichiers du répertoire */res/layout* dont le nom apparait comme argument (après ***R.layout.***) de l'appel à **setContentView** dans la méthode **onCreate** d'une activité. En effet, dans une application Android, à chaque fois qu'un appel à **setContentView** est effectué dans une activité, tous les éléments graphiques du fichier de *layout* correspondant à l'activité sont instanciés. Par exemple, si on considère le fichier de *layout* suivant (fichier de *layout* de l'application *MyHello.apk* présente dans le répertoire test de Rundroid) :

```
< ?xml version="1.0" encoding="utf-8" ?>
<LinearLayout android:orientation="vertical" android:layout_width=
        "fill_parent" android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android">
    <RelativeLayout android:id="@id/relativeLayout1"
    android:layout_width="fill_parent" android:layout_height=
    "fill_parent">
        <TextView android:id="@id/message" android:layout_width=
        "fill_parent" android:layout_height="wrap_content"
        android:text="@string/emptyString"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true" />
        <Button android:id="@id/button1" android:layout_width=
        "wrap_content" android:layout_height="wrap_content"
        android:text="@string/sayHello" android:layout_below=
        "@id/message" android:layout_alignParentLeft="true"
        android:onClick="sayHello" />
        <Button android:id="@id/button2" android:layout_width=
        "wrap_content" android:layout_height="wrap_content"
        android:layout_marginLeft="15.0dip" android:text=
        "@string/erase" android:layout_toRightOf="@id/button1"
        android:layout_below="@id/message" android:onClick="erase" />
    </RelativeLayout>
</LinearLayout>
```

Une nouvelle instance de "LinearLayout", de "RelativeLayout", de "TextView" ainsi que de "Button" est créée. Ce qui implique qu'une arête est ajoutée dans le graphe d'appels entre la méthode **setContentView** de l'activité et le constructeur (la méthode "<**init**>" en *bytecode* Dalvik) des éléments graphiques ("LinearLayout", "RelativeLayout", "TextView", ...).

Les fichiers de *layout* permettent également d'avoir des informations sur les méthodes appelées par la méthode **onClick** de la classe **android.view.View.OnClickListener**. Il suffit pour cela de regarder chaque occurrence de "*android:onClick=*" dans le fichier xml et la méthode indiquée entre guillemets à droite du égal. On trace ensuite une arête dans le graphe d'appels entre la méthode **onClick** de la classe **android.view.View.OnClickListener** et les méthodes figurant après une occurrence de "*android:onClick=*".

Le traitement des éléments graphiques instanciés et des methodes appelées par la méthode **onClick** est effectué dans l'outil Rundroid amélioré la manière suivante : à chaque fois qu'un appel à la méthode **setContentView** d'une activité est effectué, les opérations qui suivent sont executées :

- Premièrement, l'argument de **setContentView** (le nom du fichier de *layout* correspondant à l'activité précédé de "R.layout.") est récupéré. En effet, dans le *bytecode* Dalvik, cet argument n'est pas sous la forme

R.layout."nom d'un fichier de layout" mais d'une adresse hexadécimale qui est en fait l'entier pointé par R.layout."nom du fichier de layout" qui est généré dans le fichier *R.java* à la création de l'apk. Il faut donc retransformer cette adresse en un élément de la forme R.layout."nom d'un fichier de layout".
Pour ce faire, Rundroid recherche dans la classe R.layout le nom du champ (qui correspond au nom du fichier de *layout*) auquel l'adresse hexadécimale a été assignée.

- Deuxièmement, le fichier de *layout* dont le nom a été retrouvé est parsé et chaque élément graphique (TextView, Button, LinearLayout, ...) est ajouté à l'ensemble des classes instanciées (de l'algorithme RTA ou 0-CFA) ou à l'ensemble de la méthode **setContentView** (de l'algorithme XTA).
- Troisièmement, une arête est rajoutée dans le graphe d'appels entre la méthode **setContentView** et le constructeur chaque élément graphique du fichier de *layout*.
- Quatrièmement, toutes les occurrences de "*android:onClick=*" sont analysées. Une arête est tracée entre la méthode **setContentView** et la méthode **setOnClickListener** de l'élément graphique qui contient cette occurrence de "*android:onClick=*". Une arête est aussi tracée entre la méthode **onClick** et la méthode à droite de chaque occurrence de "*android:onClick=*".

Voici, ci-dessous (figures 7 et 8), le graphe d'appels du programme Test (dont le code est dans la section Annexes) qui est le même graphe d'appels qu'à la figure 6 dans lequel on a zoomé à d'autres endroits et auquel on a rajouté les éléments de *layout* (encadrés en rouge).

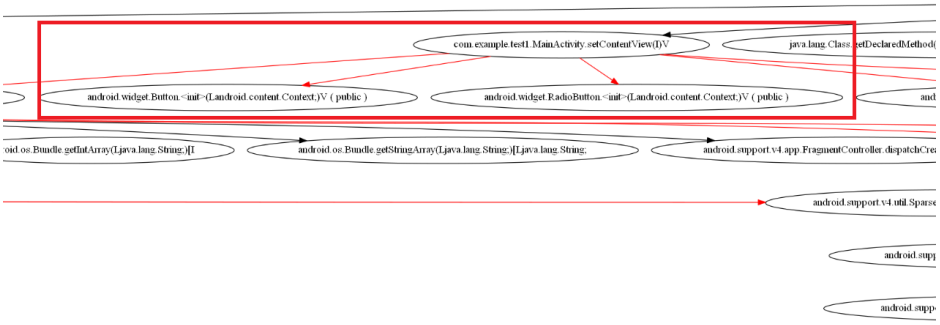

FIGURE 7 – Partie du graphe d'appels de Rundroid avec les éléments de layout

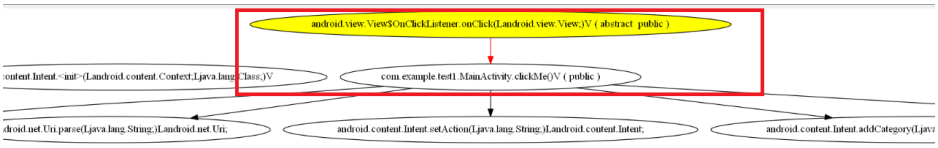

FIGURE 8 – Partie du graphe d'appels de Rundroid avec les éléments de layout

4.5 Ajout des méthodes appelées via la réflexion

L'approche pour traiter la réflexion a été de regarder chaque appel à la méthode **invoke** et de créer une arête dans le graphe d'appels entre cette méthode **invoke** et l'argument de l'appel à **getMethod** ou **getDeclaredMethod** (qui est en fait le nom de la méthode à appeler). Si nous prenons par exemple ce bout de code d'une application :

```
SecondActivity secActivityClass = new SecondActivity();
Class< ?> secActivity = null;
Method helloMethod = null;
try {
      secActivity = Class.forName("com.firstapp.SecondActivity");
      helloMethod = secActivity.getDeclaredMethod("sayHello");
      helloMethod.invoke(secActivityClass,null);
} catch (Exception e){ e.printStackTrace(); }
```

qui correspond au *bytecode* Dalvik suivant (généré avec apktool.jar : [16]) :

```
new-instance v0, Lcom/firstapp/SecondActivity;

invoke-direct {v0}, Lcom/firstapp/SecondActivity;-><init>()V

const/4 v1, 0x0

const/4 v2, 0x0

  :try_start_0
const-string v3, "com.firstapp.SecondActivity"

invoke-static {v3}, Ljava/lang/Class;->
forName(Ljava/lang/String;)Ljava/lang/Class;

move-result-object v3

move-object v1, v3

const-string v3, "sayHello"

const/4 v4, 0x0

new-array v4, v4, [Ljava/lang/Class;

invoke-virtual {v1, v3, v4}, Ljava/lang/Class;->getDeclaredMethod(
```

```
Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;

move-result-object v3

move-object v2, v3

const/4 v3, 0x0

invoke-virtual {v2, v0, v3}, Ljava/lang/reflect/Method;->
invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
  :try_end_0
.catch Ljava/lang/Exception; {  :try_start_0 ..   :try_end_0}   :catch_0

goto   :goto_0

  :catch_0
move-exception v3

invoke-virtual {v3}, Ljava/lang/Exception;->printStackTrace()V

  :goto_0
```

Nous pouvons constater qu'il y a un appel à la méthode **invoke**. Il faut donc obtenir le contenu (ici, "sayHello") du premier argument passé (le deuxième dans le *bytecode* Dalvik) à l'appel à la méthode **getDeclaredMethod** (v3) dont la valeur de retour (v3 ensuite placé dans v2) est le premier argument de la méthode **invoke** dans le *bytecode* et dont le premier argument dans le *bytecode* (v1) est la classe dont l'instance (v0) est le deuxième argument dans le *bytecode* de l'appel à **invoke**.

La procédure détaillée pour traiter la réflexion dans le programme Rundroid amélioré est donc la suivante : à chaque fois qu'un appel à **invoke** survient dans le code, les étapes suivantes sont effectuées :

1. On récupère le nom de la variable passée en premier argument de la méthode **invoke** contenant l'instance (via l'instruction "new-instance" ou un appel à **newInstance()**) de la classe de la méthode qui doit être invoquée.
2. Si l'instance de la classe de la méthode à invoquer a été créée via l'instruction "new-instance", on regarde le contenu (l'opérande de cette instruction) de la variable récupérée à la première étape afin d'obtenir le nom de la classe de la méthode qui doit être invoquée et on passe à l'étape suivante (étape 3).
   Sinon, si l'instance de cette classe a été créée via un appel à **newInstance()**, on cherche le résultat de l'appel à **newInstance()** correspondant à la variable contenant cette variable d'instance.
   Pour ce faire, on regarde les occurrences de "move-result-object" et "move-object" (car le résultat de l'instruction "move-result-object" peut être transféré dans une autre variable comme expliqué à l'étape 4) dans le code de la méthode appelant **invoke** dont la variable cible correspond à cette variable d'instance.
   On cherche ensuite la variable passée en argument de l'appel à **newInstance()** (donc l'instruction juste avant "move-result-object"). Cette variable est le résultat d'un appel à la méthode **forName** de la classe **java.lang.Class**. Il faut donc retrouver l'occurrence de "move-object"

ou "move-result-object" contenant ce résultat.
Il est ensuite aisé de remonter dans les instructions pour retrouver l'appel à **forName** ayant donné lieu à ce résultat. Dans cet appel à **forName**, il faut extraire la variable correspondant à l'argument utilisé dans ce dernier.
Enfin, on cherche l'instruction "const-string" dont la variable cible est la variable contenant l'argument de **forName**. On récupère ensuite la chaîne de caractères de cette instruction et cela nous donne le nom de la classe de la méthode à invoquer. On passe ensuite directement à l'étape 5.

3. On regarde si l'instruction "const-string" n'est pas utilisée avec comme chaîne de caractères le nom de la classe de la méthode à invoquer. Sinon, on regarde si un appel à **getClass** avec, comme premier argument dans le *bytecode*, l'instance de la classe de la méthode à appeler, ne figure pas dans le code de la méthode appelant **invoke**.
   Si on rencontre l'instruction "const-string" avec comme chaîne de caractères le nom de la classe de la méthode à invoquer, on vérifie si le code de la méthode appelant **invoke** ne contient pas par la suite un appel à **forName** et que cet appel n'a pas comme argument la variable cible du "const-string" et, si c'est le cas, on passe à l'étape suivante.
   Sinon, si on rencontre un appel à la méthode **getClass** avec, comme premier argument dans le *bytecode*, l'instance de la classe de la méthode à appeler, on passe à l'étape suivante.
4. On regarde le nom de la variable dans lequel le résultat de l'appel à **forName** ou **getClass** a été stocké (via l'instruction "move-result-object"). À noter que le contenu de cette variable peut ensuite être assigné à une autre variable via l'instruction "move-object". Il est donc important de vérifier la présence de cette instruction avec comme variable source la variable résultat de l'appel à **forName** ou **getClass**.
5. On vérifie si la variable dans laquelle le résultat de **forName** ou **getClass** est stocké (ou la variable dans laquelle ce résultat est éventuellement ensuite transféré) est présente dans l'appel à **getDeclaredMethod** ou **getMethod**.
   Si c'est le cas, on récupère le nom de la variable contenant le nom de la méthode à appeler qui est le deuxième argument (dans le *bytecode*) de cet appel à **getDeclaredMethod** ou **getMethod**.
6. On cherche dans le code de la méthode appelant **invoke** une instruction "const-string" dont la variable cible/résultat est la même que la variable passée en deuxième argument (dans le *bytecode*) de **getDeclaredMethod** ou **getMethod**.
   On regarde ensuite la chaîne de caractères associée à cette instruction "const-string" afin d'obtenir le nom de la méthode à appeler via la méthode **invoke**.
7. Il ne reste plus qu'à ajouter la classe de la méthode à invoquer dans l'ensemble des classes instanciées (de l'algorithme RTA ou 0-CFA) ou dans l'ensemble de la méthode **invoke** (de l'algorithme XTA) et à tracer une arête entre la méthode **invoke** et la méthode à invoquer via la réflexion.

Voici, ci-dessous, le graphe d'appels du programme Test qui est le même qu'aux figures 7 et 8 zoomé au même endroit que sur la figure 6 auquel on a rajouté la gestion de la réflexion (encadrée en rouge)

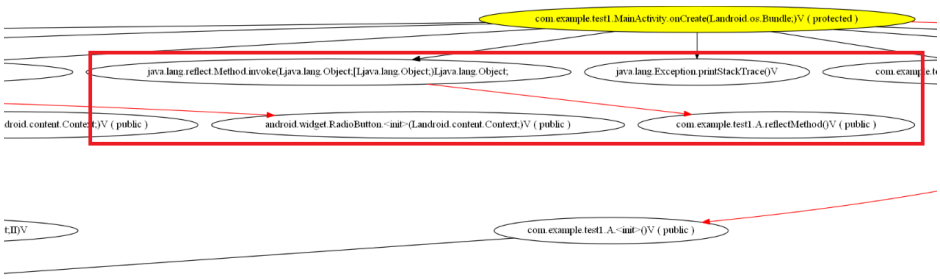

FIGURE 9 – Partie du graphe d'appels de Rundroid avec la réflexion

4.6 Ajout des intents

4.6.1 Traitement des intents explicites

L'idée de base pour traiter les *intents* explicites est de récupérer le nom de la classe passée en argument, soit sous forme de String si l'*intent* a un constructeur de la forme *Intent(String)*, soit sous forme d'un objet (de type Class) stocké dans une variable via l'instruction "const-class" si l'*intent* a un constructeur de la forme *Intent(Context, Class)* ou *Intent(String, Uri, Context, Class)*. On crée ensuite une arête entre la méthode **startActivity**, **startActivityForResult**, **startService**, **bindService**, **sendBroadcast** ou **sendOrderedBroadcast** et la méthode **onCreate** et **onStart** ou **onActivityResult** ou **onCreate** et **onStartCommand** ou **onCreate** et **onBind** ou **onReceive**. Par exemple, si on a le bout de code suivant :

```
Intent intent = new Intent(this,com.firstapp.OtherActivity.class);
startActivity(intent);
```

qui correspond au *bytecode* Dalvik suivant :

```
new-instance v3, Landroid/content/Intent;

const-class v4, Lcom/firstapp/OtherActivity;

invoke-direct {v3, p0, v4}, Landroid/content/Intent;->
<init>(Landroid/content/Context;Ljava/lang/Class;)V

invoke-virtual {p0, v3}, Lcom/firstapp/MainActivity;->
startActivity(Landroid/content/Intent;)V
```

On voit qu'il s'agit d'un *intent* avec un constructeur *Intent(Context,Class)*. Il faut donc récupérer le contenu de la variable passée en troisième argument dans le *bytecode* Dalvik du constructeur. Pour cela il suffit de chercher l'instruction "const-class" qui contient cette variable comme variable cible et de récupérer le nom de la classe de cette instruction (ici, "Lcom/firstapp/OtherActivity;").

Ensuite, on trace une arête dans le graphe d'appels entre la méthode **startActivity** et les méthodes **onCreate** et **onStart** de la classe (ici, la classe correspond à une activité).

Voici comment, dans le programme Rundroid amélioré, les *intents* explicites sont traités :

- Premièrement, on regarde si une occurrence de la méthode **startActivity**, **startActivityForResult**, **startService**, **bindService**, **sendBroadcast** ou **sendOrderedBroadcast** est présente dans le code.
  Si c'est le cas, on passe à l'étape suivante.
- Deuxièmement, on récupère la variable correspondant à l'*intent* passée en argument de cette méthode. On regarde ensuite les occurrences de "Landroid/content/Intent ;-><init>" qui contiennent cette variable en argument dans le code.
  On vérifie ensuite si cette occurrence contient "(Ljava/lang/String ;)" après "<init>". Ce qui correspond au constructeur "Intent(String)".
  Si c'est le cas, on récupère la variable passée en deuxième argument de ce constructeur dans le *bytecode* (le nom de la classe correspondant à l'activité, au service ou au BroadcastReceiver).
  Sinon, si l'occurrence contient"(Landroid/content/Context ;Ljava/lang/Class ;)" - ce qui correspond au constructeur "Intent(Context,Class)" - , on récupère la variable passée en troisième argument de ce constructeur dans le *bytecode* (la classe elle-même).
  Sinon, si l'occurrence contient "Ljava/lang/String ;Landroid/net/Uri ;Landroid/content/Context ;Ljava/lang/Class ;)" - ce qui correspond au constructeur "Intent(String,Uri,Context,Class)" - , on récupère la variable passée en cinquième argument de ce constructeur dans le *bytecode* (la classe elle-même).
- Troisièmement, si le constructeur d'*intent* découvert à l'étape précédente était de la forme "Intent(String)", on récupère le nom de la classe cible en extrayant la chaîne de caractères dans l'occurrence de "const-string" contenant la variable récupérée à l'étape précédente comme variable cible.
  Sinon si le constructeur était de la forme "Intent(Context,Class)" ou "Intent(String,Uri,Context,Class)", on récupère la classe cible en extrayant la classe dans l'occurrence de "const-class" contenant la variable récupérée à l'étape précédente comme variable cible.
- Quatrièmement, si, à la première étape, un appel à la méthode **startActivity** a été détecté, une arête est ajoutée entre cette méthode **startActivity** et les méthodes **onCreate** et **onStart** de la classe (ou le nom de la classe) qui a été récupérée à l'étape précédente (ici, c'est une activité).
  Sinon, si c'est un appel à **startActivityForResult** qui a été détecté, une arête est ajoutée entre cette méthode et la méthode **onActivityResult** de la classe récupérée à l'étape précédente.
  Sinon, si c'est un appel à **startService** qui a été détecté, une arête est ajoutée entre cette méthode et les méthodes **onCreate** et **onStartCommand** de la classe (ici, c'est un service) récupérée à l'étape précédente.
  Sinon, si c'est un appel à **bindService** qui a été détecté, une arête est ajoutée entre cette méthode et les méthodes **onCreate** et **onBind** de la classe récupérée à l'étape précédente.

Sinon, si c'est un appel à **sendBroadcast** ou **sendOrderedBroadcast** qui a été détecté, une arête est ajoutée entre cette méthode et la méthode **onReceive** de la classe (ici, c'est un BroadcastReceiver) récupérée à l'étape précédente.

4.6.2 Traitement des intents implicites

La définition des *intents* donnée en introduction indique que les *intents* implicites se voient attribuer une description d'une classe plutôt qu'une classe elle-même. Pour que l'*intent* reçoive la description la plus précise possible de la classe ou des classes (de l'activité ou des BroadcastReceivers) à activer, il faut lui fournir au moins une des trois caractéristiques suivantes de la classe : le(s) nom(s) de l'action ou des actions acceptée(s) par la classe, les données à fournir à la classe et le nom de la catégorie ou des catégories acceptée(s) par la classe.

Parmi les types d'actions courantes, nous pouvons citer ACTION_VIEW, utilisée quand une activité possède une information qu'elle peut montrer à un utilisateur comme une photo à visualiser dans la galerie ou une adresse à visualiser dans une application comme Google Maps. Nous pouvons aussi citer ACTION_SEND, qui est utilisée, quant à elle, lorsque qu'une application possède des données que l'utilisateur peut partager avec une autre application ou activité comme l'email ou une application de réseau social.

Concernant les données, ces dernières dépendent du type d'action précisé. Par exemple, si l'action est de type ACTION_SEND, la donnée fournie peut être l'URI d'un fichier à envoyer ou simplement du texte. Par contre, si l'action est de type ACTION_EDIT (quand une application peut accéder à un fichier en écriture), la donnée fournie doit obligatoirement être l'URI du fichier à modifier.

Pour les catégories, parmi celles qui sont les plus utilisées, nous pouvons citer la catégorie CATEGORY_BROWSABLE qui indique que l'activité cible peut être lancée par un navigateur web pour afficher des données comme un lien, une image ou un email. Nous pouvons aussi citer la catégorie CATEGORY_LAUNCHER qui indique que l'activité est une activité lancée en premier et que cette activité figure dans la liste du *launcher* d'applications (équivalent du bureau sous Android).

Les actions sont fournies à l'*intent* via la méthode **setAction**, les données, elles, sont fournies à l'*intent* via la méthode **setData** et les catégories via la méthode **addCategory**. Ces trois méthodes font partie de la classe Intent.

Lorsque l'*intent* va ensuite être exécuté via la méthode **startActivity**, **startActivityForResult**, **sendBroadcast** ou **sendOrderedBroadcast**, l'application va chercher dans le fichier *AndroidManifest.xml* la partie "intent-filter" dont les sous-parties "action", "data" et "category" se rapprochent le plus des informations données, respectivement, par **setAction**, **setData** et **addCategory**. Il va ensuite lancer l'activité ou le BroadcastReceiver contenant cet

"intent-filter". Pour la partie "data", il est suffisant que le préfixe de la donnée fournie par **setData** y figure pour que l'activité ou le BroadcastReceiver contenant cet "intent-filter" soit lancé(e).

Voici un exemple d'*intent* implicite et de comment les trois méthodes mentionnées ci-dessus peuvent être utilisées :

```
final String request = "https://www.google.fr/";
Intent intent = new Intent();
intent.setAction(Intent.ACTION_VIEW);
intent.setData(Uri.parse(request));
intent.addCategory(Intent.CATEGORY_BROWSABLE);
startActivity(intent);
```

Ce code Java correspond au *bytecode* Dalvik suivant :

```
const-string v0, "https://www.google.fr/"

new-instance v1, Landroid/content/Intent;

invoke-direct {v1}, Landroid/content/Intent;-><init>()V

const-string v2, "android.intent.action.VIEW"

invoke-virtual {v1, v2}, Landroid/content/Intent;->
setAction(Ljava/lang/String;)Landroid/content/Intent;

const-string v2, "https://www.google.fr/"

invoke-static {v2}, Landroid/net/Uri;
->parse(Ljava/lang/String;)Landroid/net/Uri;

move-result-object v2

invoke-virtual {v1, v2}, Landroid/content/Intent;->
setData(Landroid/net/Uri;)Landroid/content/Intent;

const-string v2, "android.intent.category.BROWSABLE"

invoke-virtual {v1, v2}, Landroid/content/Intent;->
addCategory(Ljava/lang/String;)Landroid/content/Intent;

invoke-virtual {p0, v1}, Lcom/firstapp/SecondActivity;->
startActivity(Landroid/content/Intent;)V
```

Dans le contexte du programme Rundroid amélioré, le traitement des *intents* implicites se fait en examinant les trois caractéristiques (actions, données et catégories) mentionnées dans le paragraphe précédent. Le fichier *AndroidManifest.xml* est aussi inspecté afin d'extraire le contenu des sous-parties "action", "data" et "category" (ces sous-parties ne figurent pas toujours toutes dans un "intent-filter") de chaque partie "intent-filter" et si les sous-parties présentes sont égales au caractéristiques fournies à l'*intent*, une arête est créée entre **startActivity**, **startActivityForResult**, **sendBroadcast** ou **sendOrderedBroadcast** et les méthodes **onCreate** et **onStart** ou **onActivityResult** ou **onReceive** de l'activité (dans la Section "activity") ou du BroadcastReceiver (dans la Section "receiver") contenant l'"intent-filter" qui comporte lui-même les caractéristiques fournies à l'*intent* dans ses sous-parties.

Si nous reprenons l'exemple du code ci-dessus, si le contenu du fichier *AndroidManifest.xml* est le suivant :

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 android:compileSdkVersion="28" android:compileSdkVersionCodename="9"
 package="com.firstapp" platformBuildVersionCode="28"
 platformBuildVersionName="9">
    <uses-permission android:name="android.permission.INTERNET"/>
    <application android:allowBackup="true"
    android:appComponentFactory="android.support.v4.app.CoreComponentFactory"
    android:debuggable="true" android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true" android:theme="@style/AppTheme">
        <activity android:name="com.firstapp.OtherActivity">
            <intent-filter>
                <action android:name="android.intent.action.VIEW"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <category android:name="android.intent.category.BROWSABLE"/>
                <data android:scheme="http"/>
                <data android:scheme="https"/>
            </intent-filter>
        </activity>
        <activity android:name="com.firstapp.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Les arguments des méthodes **setAction**, **setData** et **addCategory** (respectivement, "android.intent.action.VIEW", "https ://www.google.fr/", "android.intent.category.BROWSABLE") sont comparés avec le contenu de chaque Section *intent-filter* du fichier *AndroidManifest.xml*. On remarque que dans la Section "intent-filter" de l'activité "OtherActivity", il y a une Sous-section "action", deux Sous-sections "category" et deux Sous-sections "data".

La Sous-section "action" contient "android.intent.action.VIEW", ce qui est également l'action passée en argument de **setAction**.

Quant aux deux Sous-sections "category", l'une contient "android.intent.category.DEFAULT" et l'autre "android.intent.category.BROWSABLE". Cette dernière catégorie est la même qui est passée en argument de **addCategory**.

Concernant les deux Sous-sections "data", l'une contient "http" et l'autre "https". Or, la donnée passée en argument de **setData** contient "https".

Une arête sera tracée entre la méthode **startActivity** de l'activité "MainActivity" et les méthodes **onCreate** et **onStart** de l'activité "OtherActivity".

En regardant la Section "intent-filter" de l'activité "MainActivity", on peut directement remarquer que l'action de la Sous-section "action" ne correspond

pas à l'argument passé à la méthode **setAction**. Cette activité n'est donc pas une cible de l'*intent* passé en argument de la méthode **startActivity**.

Si l'*intent* a une signature du type "Intent ;-><init>(Ljava/lang/String ;Landroid/net/Uri ;)V", si l'URI passée en argument (le troisième argument dans le *bytecode* Dalvik) contient "http" ou "geo :", la procédure sera différente.

Si l'URI contient "http", si l'*intent* est utilisé comme argument de la méthode **startActivity**, une arête est ajoutée dans le graphe d'appels entre cette méthode **startActivity** et les méthodes **onCreate** et **onStart** des activités **com.google.android.apps.chrome.Main**, **www.ijoysoft.browser.activities.MainActivity** et **org.mozilla.firefox.App**. Ce choix est justifié par le fait que les trois navigateurs les plus utilisés sous Android sont Chrome, Safari et Firefox [5] et que les activités principales de ces navigateurs, sont, respectivement, **android.apps.chrome.Main**, **www.ijoysoft.browser.activities.MainActivity** et **org.mozilla.firefox.App**.

Pour trouver les activités principales (les activités qui se lancent au démarrage d'une application) des trois applications de navigation citées ci-dessus, le graphe d'appels de ces trois applications a été construit avec l'outil Rundroid amélioré car ce dernier colore les méthodes qui sont des racines ou points d'entrée de l'application en jaune (cf. Sous-section 4.3).

Si l'URI contient "geo :", cela signifie que l'activité cible de l'*intent* devra être capable de gérer des coordonnées géographiques. De ce fait, si l'*intent* est utilisé comme argument de la méthode **startActivity**, une arête est ajoutée dans le graphe d'appels entre cette méthode **startActivity** et les méthodes **onCreate** et **onStart** de l'activité **com.google.android.gms.maps.MapFragment**. Cela est dû au fait que cette dernière est l'activité principale de l'application Google Maps qui est l'application la plus utilisée de géolocalisation et de guidage GPS [6] et est disponible nativement sous Android. L'activité principale a été trouvée, encore une fois, grâce à la construction du graphe d'appels via l'outil Rundroid amélioré.

La procédure détaillée du traitement des *intents* implicites dans le contexte de l'outil Rundroid amélioré est la suivante :

- Premièrement, on regarde si une occurrence de la méthode **startActivity**, **startActivityForResult**, **startService**, **bindService**, **sendBroadcast** ou **sendOrderedBroadcast** est présente dans le code.
  Si c'est le cas, on passe à l'étape suivante.
- Deuxièmement, on récupère la variable correspondant à l'*intent* passée en argument de cette méthode. On regarde ensuite les occurrences de "Landroid/content/Intent ;-><init>" qui contiennent cette variable en argument dans le code.
  On vérifie ensuite si cette occurrence contient "(Ljava/lang/String ;)" après "<init>" . Ce qui correspond au constructeur "Intent(String)".

5. https ://en.wikipedia.org/wiki/Usage_share_of_web_browsers
6. https ://blog.sagipl.com/most-used-apps/#Google_Maps

Si c'est le cas, on récupère la variable passée en deuxième argument de ce constructeur dans le *bytecode* (le nom de l'action servant à décrire la ou les activités cible(s) ou le BroadcastReceiver cible de l'*intent*).
De plus, si une occurrence de la méthode **setData** de la classe Intent avec comme premier argument dans le *bytecode* la variable correspondant à celle récupérée à la première étape est trouvée, on récupère la deuxième variable passée en argument de cette méthode (la donnée fournie à l'*intent*).
Aussi, si une occurrence de la méthode **addCategory** (aussi de la classe Intent) avec comme premier argument dans le *bytecode* la variable correspondant à celle récupérée à la première étape est trouvée, on récupère la deuxième variable passée en argument de cette méthode (la catégorie servant à décrire la classe cible).
Sinon, si l'occurrence contient "(Ljava/lang/String ;Landroid/net/Uri ;)" - ce qui correspond au constructeur "Intent(String,Uri)" -, on récupère la deuxième et la troisième variable passées en argument de ce constructeur dans le *bytecode* (respectivement, le nom de l'action décrivant la classe cible et l'URI qui est la donnée à traiter par la classe cible).
Sinon, si l'occurrence contient juste () après "< init>" - ce qui correspond au constructeur "Intent()" -, on regarde les occurrences des méthodes **setAction**, **setData** et **addCategory** dans la méthode trouvée à la première étape.
Si on trouve une des ces méthodes avec, en argument, la variable trouvée à la première étape, on récupère la variable passée en deuxième argument dans le *bytecode* de cette méthode.

- Troisièmement, si le constructeur de l'*intent* était du type "Intent(String)", on regarde si une occurrence de "const-string" dont la variable cible est égale à la variable correspondant à l'action récupérée à l'étape précédente et dont la chaîne de caractères contient "intent.action" est présente dans le code de la méthode appelant la méthode trouvée à la première étape.
Si c'est le cas, on récupère le contenu de la chaîne de caractères de l'instruction "const-string".
Sinon, si le constructeur de l'*intent* était du type "Intent(String,Uri)", on vérifie si le code de la méthode appelant la méthode trouvée à la première étape contient l'instruction "const-string" et que cette instruction possède la variable contenant l'action récupérée à l'étape précédente comme variable cible.
Si c'est le cas, on récupère le contenu de la chaîne de caractères de cette instruction.
On regarde également si une occurrence de "const-string" dont la variable cible est la variable contenant l'URI à parser passée en argument de la méthode **parse** de la classe **android.net.Uri** dont le résultat (de **parse**) est la variable contenant l'Uri récupérée à l'étape précédente est présente dans la méthode.
Si c'est le cas, on récupère aussi le contenu de la chaîne de caractères de l'instruction "const-string".
Sinon, si le constructeur était du type "Intent()", on regarde si il existe des occurrences de "const-string" dont la variable cible est la variable contenant soit l'action, soit la donnée, soit la catégorie décrivant la classe cible récupérée à l'étape précédente. Si c'est le cas, on récupère le contenu

de la chaîne de caractères de cette instruction.

- Quatrièmement, tous les éléments figurants dans les Sections "intent-filter" (ce qui se trouve dans chaque Sous-section "action", "category" et "data") du fichier *AndroidManifest.xml* de l'application à analyser sont extraits de ce dernier.
  Les chaînes de caractères contenant l'action, les données et la catégorie récupérées à l'étape précédente sont ensuite comparées avec les éléments extraits au début de l'étape.
  On vérifie si tous les noms des actions et des catégories récupérés à l'étape précédente sont identiques aux éléments dans les Sous-sections "action" et "category" d'une Section "intent-filter" et si les éléments dans la Sous-section "data" sont inclus dans les chaînes de caractères correspondants aux données récupérées à l'étape précédente.
  Dans ce cas là, on regarde le nom de l'activité (dans la Section "activity") ou du BroadcastReceiver (dans la Section "receiver") et on relie dans le graphe d'appels la méthode récupérée à la première étape (**startActivity**, **startActivityForResult**, **sendBroadcast** ou **sendOrderedBroadcast**) aux méthodes **onCreate** et **onStart** ou à la méthode **onActivityResult** de cette activité ou à la méthode **onReceive** de ce BroadcastReceiver.
  Dans le cas où l'URI (si elle existe) récupérée à l'étape précédente contient "http" ou "geo :", comme précisé plus haut, la procédure est différente. Cette dernière est explicitée trois pages plus haut.

Voici, ci-dessous, le graphe du programme Test auquel on a rajouté les *intents* explicites et implicites (ici, les deux types d'*intent* appelle la même activité). Il s'agit du même graphe qu'à la figure 9 dans lequel on a zoomé à un autre endroit. La gestion de l' *intent* est encadrée en rouge sur le graphe. Le graphe ci-dessous possède donc toutes les caractéristiques d'Android.

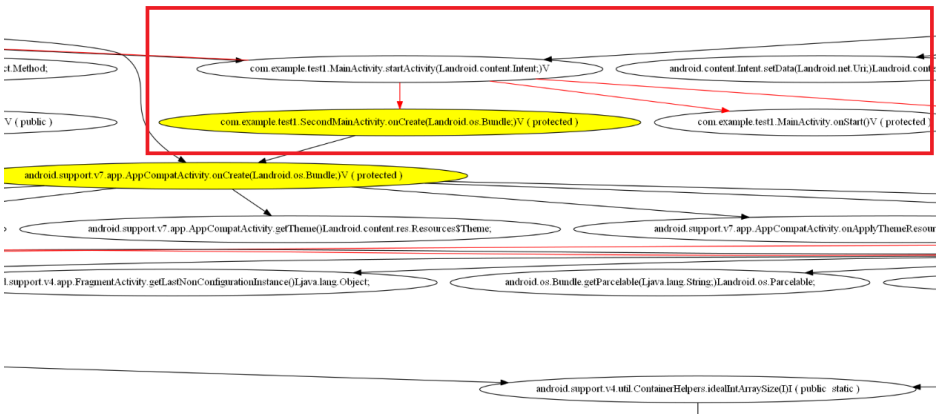

FIGURE 10 – Partie du graphe d'appels de Rundroid complet

4.7 Ajout de l'algorithme XTA

Dans l'outil Rundroid amélioré, l'algorithme XTA est implémenté de la manière suivante : trois ensembles (stockés dans des variables) principaux sont utilisés :

- un ensemble contenant les méthodes atteignables
- un ensemble contenant tous les ensembles des méthodes (une *HashMap* reliant chaque méthode à son ensemble)
- un ensemble contenant tous les ensembles des champs (une *HashMap* reliant chaque champ à son ensemble).

1. Au début de l'algorithme, les points d'entrée (cf. 4.2) sont ajoutés à la variable contenant l'ensemble des méthodes atteignables (qui était auparavant vide).
2. Si une instruction "new-instance" est rencontrée dans une méthode atteignable, l'opérande de cette instruction (la classe à instancier) est ajoutée, sous forme de *String* à l'ensemble de cette méthode atteignable.
   Si cette méthode figure déjà comme clé de la *HashMap* contenant toutes les méthodes et leur ensemble, l'ensemble correspondant à la méthode est juste étendu avec la classe instanciée.
   Sinon, la méthode atteignable est ajoutée comme clé de la *HashMap* et on lui associe comme valeur un ensemble contenant la classe instanciée.
3. Si une instruction "iget" ou "iget-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char") est trouvée dans une méthode atteignable, on vérifie si l'ensemble correspondant à l'opérande de cette instruction (le champ à lire) existe (donc si le champ est une clé de la *HashMap* contenant les champs et leur ensemble).
   Si c'est le cas, on ajoute le contenu de l'ensemble du champ à lire à l'ensemble de la méthode atteignable, si cette dernière est une clé de la *HashMap* contenant les méthodes et leur ensemble.
   Si elle n'est pas une clé, on la rajoute comme clé et on lui associe comme valeur un ensemble constitué du contenu de l'ensemble du champ à lire.
4. Si une instruction "iput" ou "iput-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char") est rencontrée dans une méthode atteignable, on regarde le type de l'opérande de cette instruction (le champ dans lequel on va écrire) ainsi que les sous-types.
   On rajoute ensuite le type et les sous-types du champ qui sont déjà dans l'ensemble de la méthode atteignable (si cette dernière est une clé de la *HashMap* contenant les méthodes et leur ensemble) à l'ensemble du champ si il est une clé de la *HashMap* contenant les champs et leur ensemble). Si il n'est pas une clé, on le rajoute comme clé et on lui associe comme valeur un ensemble contenant son type et ses sous-types déjà dans l'ensemble de la méthode atteignable.
5. Si une instruction "sget" ou "sget-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char") est rencontrée dans une méthode atteignable, on vérifie que l'opérande de l'instruction est un champ.
   Si c'est le cas, la procédure est la même que pour "iget" ou "iget-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char").
6. Si une instruction "sput" ou "sput-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char") est rencontrée dans une méthode atteignable, on vérifie également que l'opérande de l'instruction est un

champ.
Si c'est le cas, la procédure est la même que pour "iput" ou "iput-kind" (où kind peut être "short", "wide", "object", "boolean", "byte" ou "char").

7. Si une instruction "invoke-kind" (où kind, ici, peut être "super", "direct", "virtual", "static" ou "interface") est rencontrée dans une méthode atteignable, on regarde la classe de la méthode appelée (l'opérande de l'instruction) avec cette instruction.
   Appelons S l'ensemble contenant cette classe et ses sous-classes.
   Pour chaque classe C dans S, on regarde, si elle fait partie de l'ensemble de la méthode appelante (si il existe).
   Si c'est le cas, on ajoute une arête entre la méthode appelante et la méthode **m()** ayant la même signature que la méthode appelée et dont la classe est C. **m()** est aussi ajoutée à l'ensemble des méthodes atteignables si elle n'y figure pas.
   Comme pour l'algorithme RTA, **m()** ne sera prise en compte que si elle est déclarée explicitement dans le code et qu'elle n'existe pas juste implicitement par une relation d'héritage (cf. 4.3).
   De plus, le type et les sous-types des arguments de **m()** déjà dans l'ensemble de la méthode appelante (si elle figure comme clé de la *HashMap* contenant les méthodes et leur ensemble) sont ajoutés à l'ensemble de **m()**.
   Si **m()** n'est pas une clé de la *HashMap* contenant les méthodes et leur ensemble, elle est ajoutée comme clé et on lui associe comme valeur un ensemble contenant le type et les sous-types de ses arguments déjà dans l'ensemble de la méthode appelante.
   Ensuite, le type et les sous-types statiques de la valeur de retour de **m()** déjà dans l'ensemble de cette méthode sont ajoutés à l'ensemble de la méthode appelante.
   Si la méthode appelante n'est pas une clé de la *HashMap* contenant les méthodes et leur ensemble, elle est ajoutée comme clé et on lui associe comme valeur un ensemble contenant le type et les sous-types du type de **m()** déjà dans l'ensemble de **m()**.
   Enfin, chaque classe dans l'ensemble de la méthode appelante (si il existe) est ajouté à l'ensemble de **m()**.
   Si **m()** n'est pas une clé de la *HashMap* contenant les méthodes et leur ensemble, elle est ajoutée comme clé et un ensemble constitué du contenu de l'ensemble de la méthode appelante lui est attribué comme valeur.
8. Toutes ces opérations sont répétées jusqu'à ce que le contenu des ensembles des méthodes et des champs ne change plus.

Dans le contexte de ce travail, l'algorithme XTA pour le *bytecode* Dalvik peut être formalisé de la manière suivante :

```
soit R = {} ensemble des méthodes atteignables.
soit H1 = {} ensemble contenant chaque méthode associée à son
ensemble.
soit H2 = {} ensemble contenant chaque champ associé à son ensemble.
soit S.Lx/y/B;->m() = {} et S.Lx/y/B;->f:Lw/z/A; = {},
respectivement, ensemble de la méthode m() de la classe Lx/y/B;
(valeur de la clé "Lx/y/B;->m()" dans H1)
et ensemble du field f, de la classe Lx/y/B; et de type Lw/z/A;
(valeur de la clé "Lx/y/B;->f:Lw/z/A;" dans H2).
```

```
A -> B signifie ajout de A dans B.
A ->> B signifie arête ajoutée entre A et B.
sous-types(c) désigne type(c) uni à tous les sous-types de c.
"kind" = "", "wide", "object", "boolean", "byte", "char" ou "short".
"kind2" = "super", "direct", "virtual", "interface" ou "static".

public static main([L/java/lang/String;)V -> R

pour chaque Lx/y/B;->m() dans R :
        pour chaque new-instance vA La/b/C; dans Lx/y/B;->m() :
                La/b/C; -> S.Lx/y/B;->m()
        pour chaque sget-kind (iget-kind) vA, (vB), La/b/C;->f:Lw/z/D;
dans Lx/y/B;->m() :
                pour chaque Cl; dans S.La/b/C;->f:Lw/z/D; :
                        Cl; -> S.Lx/y/B;->m()
        pour chaque  sput-kind (iput-kind) vA, (vB), La/b/C;->f:Lw/z/D;
        dans Lx/y/B;->m() :
                pour chaque Cl; dans sous-types(La/b/C;->f:Lw/z/D) :
                        si Cl; dans S.Lx/y/B;->m() :
                                Cl; -> S.La/b/C;->f:Lw/z/D;
        pour chaque invoke-kind2 {vA, vB, ...} Lw/z/D;->n()
        dans Lx/y/B;->m() :
                pour chaque C; dans sous-types(Lw/z/D;) :
                        si C; dans S.Lx/y/B;->m() :
                                C;->n() -> R
                                Lx/y/B;->m() ->> C;->n()
                                C; -> S.C;->n()
                                pour chaque A dans
                                sous-types(arguments(C;->n()) :
                                        si A dans S.Lx/y/B;->m() :
                                                A -> S.C;->n()
                                pour chaque N dans
                                sous-types(retour(C;->n()) :
                                        si N dans S.C;->n() :
                                                N -> S.Lx/y/B;->m()
```

Avant l'affichage du graphe d'appels, l'ensemble de chaque et de chaque champ est affiché dans la console (voir figure 11). Si le nom d'une méthode ou d'un champ du programme ne figure pas dans ces ensembles affichés, c'est l'ensemble correspondant à cette méthode ou ce champ est vide.

```
Methods sets:
-------------
classes of Lcom/example/testrtaapp/A;->public m2()V = { 1893 (Lcom/example/testrtaapp/B;) 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/C;->public m4()V = { 1893 (Lcom/example/testrtaapp/B;) 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/Main;->public static main([Ljava/lang/String;)V = { 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/B;->public constructor <init>()V = { 1893 (Lcom/example/testrtaapp/B;) 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/B;->public m3()V = { 1893 (Lcom/example/testrtaapp/B;) 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/A;->public constructor <init>()V = { 1892 (Lcom/example/testrtaapp/A;) 1895 (Lcom/example/testrtaapp/C;) }
classes of Lcom/example/testrtaapp/B;->constructor static <clinit>()V = { 1892 (Lcom/example/testrtaapp/A;) }
classes of Lcom/example/testrtaapp/C;->public constructor <init>()V = { 1895 (Lcom/example/testrtaapp/C;) }
classes of Ljava/lang/Object;-><init>()V = { 1895 (Lcom/example/testrtaapp/C;) }
Fields sets:
------------
classes of Lcom/example/testrtaapp/B;.f = { 1892 (Lcom/example/testrtaapp/A;) }
```

FIGURE 11 – Affichage des ensembles des méthodes et des fields

4.8 Ajout de l'algorithme 0-CFA

L'algorithme 0-CFA est implementé de la manière suivante dans l'outil Rundroid amélioré : La procédure de construction du graphe d'appels lui-même est la même que pour l'algorithme RTA. Cependant, une analyse du type de chaque

variable est effectuée en parallèle de la construction du graphe d'appels. Pour cette analyse, quatre ensembles (variables) sont utilisés :

- un cache (*HashMap*) contenant toutes les méthodes rencontrées indexées par leur nom
- un cache contenant tous les sites d'appels (méthodes appelées) rencontrés indexés par leur nom
- un cache contenant tous les types effectifs des arguments et les types effectifs de retour indexés par le nom de la méthode correspondante
- un cache contenant les types effectifs des variables (qui sont en réalité des registres) indexés par le nom des variables

Le nom de ces variables, dans l'implémentation, sera le nom de la classe de la variable suivi du nom de la méthode de la variable suivi du nom de la variable lui-même.
Exemple : *Lcom.app.MainActivity ;m()Vv4*

1. À chaque fois que l'instruction "move-object" ou "move-object/from16" est rencontrée, on regarde si le nom de la variable source (ou variable opérande) de l'instruction fait partie des clés du cache des types effectifs des variables.
   Si c'est le cas, la valeur (le type) associé à la variable source est ajouté aux types de la variable cible de l'instruction.
   Si le nom de la variable cible ne fait pas partie des clés du cache des types effectifs des variables, le nom de cette variable est ajouté comme clé et un ensemble contenant le type de la variable source lui est associé comme valeur.
2. À chaque fois que l'instruction "move-result-object" est rencontrée, on cherche la variable retournée par la méthode de l'instruction précédente (si il s'agissait "invoke-kind") ou son opérande (si il s'agissait de l'instruction "filled-new-array" ou "filled-new-array-range").
   Si l'instruction précédente était "invoke-kind" (ou kind peut être *super*, *direct*, *virtual*, *static* ou *interface*), on regarde si la variable retournée par la méthode appelée fait partie du cache contenant les variables et leurs types.
   Si c'est le cas, on ajoute les types de la variable retournée à l'ensemble des types de la variable cible de l'instruction "move-result-object". Si cette dernière ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant les types de la variable retournée par la méthode appelée par l'instruction précédente.
   Sinon, si l'instruction précédente était "filled-new-array" ou "filled-new-array-range", on récupère l'opérande de cette instruction (qui est en fait le type du tableau) et l'ajoute à l'ensemble des types de la variable cible de "move-result-object".
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant le type du tableau.
3. Si une occurrence de l'instruction "const-string" est rencontrée dans une méthode atteignable, le type "Ljava/lang/String ;" est ajouté à l'ensemble des types de la variable cible de cette instruction.
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un

ensemble contenant l'élément "Ljava/lang/String;".

4. Si l'instruction "const-class" ou "new-instance" est rencontrée dans une méthode atteignable, on ajoute l'opérande de cette instruction (la classe) à l'ensemble des types de la variable cible de l'instruction.
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant l'opérande de l'instruction (la classe à mettre dans la variable cible ou la classe dont l'instance est à mettre dans cette variable).
5. Si l'instruction "new-array" est rencontrée dans une méthode atteignable, on ajoute également l'opérande de cette instruction (le type du tableau vide) à l'ensemble des types de la variable cible de l'instruction.
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant le type du nouveau tableau vide.
6. Si l'instruction "aget-object" est rencontrée dans une méthode atteignable, on vérifie si le deuxième argument de l'instruction (la deuxième variable) qui est le tableau dont on veut extraire un élément fait partie du cache contenant les variables et leurs types.
   Si c'est le cas, on regarde les types de cette variable dans le cache et on les ajoute à l'ensemble des types de la variable cible en leur retirant le symbole "[".
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant les types de la variable cible sans le symbole "[".
7. Si l'instruction "iget-object" est rencontrée dans une méthode atteignable, on récupère le type de l'opérande de l'instruction (le champ à lire).
   On ajoute ensuite le type du champ récupéré à l'ensemble des types de la variable cible.
   Si la variable cible ne fait pas partie du cache contenant les variables et leurs types, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant le type du champ à lire.
8. Si l'instruction "sget-object" est recontrée dans une méthode atteignable, on vérifie si l'opérande de l'instruction est bien un champ.
   Si c'est le cas, la procédure est la même que pour l'instruction "iget-object".
9. À chaque fois qu'une instruction "invoke-kind" (où kind peut être *super*, *direct*, *virtual*, *static* ou *interface*) est rencontrée, on extrait les variables correspondant aux arguments et on ajoute le nom complet de ces variables à l'ensemble des arguments de la méthode appelée (qui est donc un site d'appel).
   On regarde ensuite si la méthode invoquée par l'instruction fait partie du cache contenant les sites d'appels, si ce n'est pas le cas, on l'ajoute dans le cache.
   Sinon, si elle en fait partie, on regarde si la méthode est appelée avec de nouveaux arguments.
   Si c'est le cas, on extrait les variables correspondant à ces nouveaux arguments et on regarde si elles font partie du cache contenant les variables et leurs types.
   Si elles en font partie, on regarde les types de ces variables et on les ajoute à l'ensemble des types des arguments et de la valeur de retour de

la méthode appelée.
Si la méthode appelée ne fait pas partie du cache contenant les méthodes et les types de leurs arguments et de leur valeur de retour, on l'ajoute comme clé on lui associe comme valeur un ensemble contenant un autre ensemble contenant les types des variables correspondant aux arguments.

10. Après l'analyse des instructions dans une méthode atteignable, on vérifie si son nom fait partie des clés du cache des méthodes rencontrées.
Si ce n'est pas le cas, on regarde si la méthode fait partie du cache contenant les sites d'appels.
Si elle en fait partie, on extrait les variables correspondant aux arguments de cette méthode de ce site d'appel.
Pour chaque variable extraite de cette manière, on vérifie ensuite si elle fait partie du cache contenant les variables et leurs types.
Si c'est le cas, on regarde le type de la variable dans ce cache (la valeur correspondante). On ajoute ensuite le type à l'ensemble des types des arguments et de la valeur de retour de la méthode analysée.
Si cette dernière ne fait pas partie du cache des méthodes et de leurs types des arguments et de la valeur de retour, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant un ensemble composé des types des arguments de la méthode.
Ensuite, on extrait le type de la valeur de retour de la méthode.
Pour ce faire, on cherche d'abord la variable correspondant à la valeur de retour de la méthode.
On regarde ensuite si cette variable fait partie du cache des variables et de leurs types.
Si c'est le cas, on regarde le type de cette variable dans le cache et on ajoute ce dernier à l'ensemble des types des arguments et de la valeur de retour de la méthode analysée.
Si la méthode analysée ne fait pas partie du cache des méthodes et de leurs types des arguments et de la valeur de retour, on l'ajoute comme clé et on lui associe comme valeur un ensemble contenant un ensemble composé des types de la valeur de retour.

La partie de calcul des types de l'algorithme 0-CFA, dans le contexte de ce travail, peut être formalisée de cette manière :

```
soit R = {}, ensemble des méthodes atteignables.
soit H1 = {}, H2 = {}, respectivement, ensemble des méthodes
rencontrées indexées par leur nom et ensemble
des méthodes appelées indexées par leur nom.
soit H3 = {}, H4 = {}, respectivement, ensemble des méthodes
et leur types effectifs des arguments et de leur valeur de
retour et ensemble des variables et leur type effectif.
clés(H) = ensemble des clés de H.
put(H,e,v) = ajout ou modification de la clé e dans H en lui
associant {v} comme valeur ou en ajoutant v au set
correspondant à sa valeur.
get(H,e) = obtention de la valeur de la clé e dans H.
type(C) = type de C.
ret(La/b/C;->m()) = valeur (variable) retournée par la méthode m()
de la classe La/b/C;.
prec(I) = instruction précédant I, une instruction.
"kind" = "super", "direct", "virtual", "static" ou "interface".
"abc" moins "a" retire la lettre "a" de "abc" si elle y figure.
```

```
args(La/b/C;->m()) = ensemble des arguments de la méthode m()
de la classe La/b/C;
A -> B signifie ajout de A dans B.
method(La/b/C;->m()) = objet méthode correspondant au nom
La/b/C;->m().

pour chaque Lx/y/C;->m() dans R :
        pour chaque move-object(/from16) vA(A), vB(BBB)
        dans Lx/y/C;->m() :
                si vB(BBB) dans clés(H4) :
                        put(H4,vA(A),get(H4,vB(BBB))
        pour chaque move-result-object vA dans Lx/y/C;->m() :
                si prec(move-result-object vA) =
                invoke-kind {vB, vC, ...} Lw/y/D;->n() :
                        si ret(Lw/y/D;->n()) dans clés(H4) :
                                put(H4,vA,get(H4,ret(Lw/y/D;->n())))
                sinon si prec(move-result-object vA) =
                filled-new-array(-range) {vB, vC, ...}, Lw/y/D; :
                        put(H4,vA,Lw/y/D;)
        pour chaque const-string vA, "string" dans Lx/y/C;->m() :
                put(H4,vA,Ljava/lang/String;)
        pour chaque const-class vA, class ou new-instance vA, class
        dans Lx/y/C;->m() :
                put(H4,vA,class)
        pour chaque new-array vA, vB, [Lw/y/D; dans Lx/y/C;->m() :
                put(H4,vA,[Lw/y/D;)
        pour chaque aget-object vA, vB, vC dans Lx/y/C;->m() :
                si vB dans clés(H4) :
                        put(H4,vA,get(H4,vB) moins "[")
        pour chaque sget-object (iget-object) vA (,vB),
        Lw/z/D;->f:La/b/C; dans Lx/y/C;->m() :
                put(H4,vA,La/b/C;)
        pour chaque invoke-kind {vA, vB, ...} Lw/z/D;->n()
        dans Lx/y/C;->m() :
                Lw/z/D;->n()vA -> args(Lw/z/D;->n())
                Lw/z/D;->n()vB -> args(Lw/z/D;->n())
                  ...
                put(H2,Lw/z/D;->n(),method(Lw/z/D;->n()))
                si args(Lw/z/D;) a changé :
                        pour chaque nouvelle variable v
                        dans args(Lw/z/D;) :
                                si v dans clés(H4) :
                                        put(H3,Lw/Z/D;->n(),
                                        {get(H4,v),/})
        si Lx/y/C;->m() pas dans clés(H1) :
                si Lx/y/C;->m() dans clés(H2) :
                        pour chaque v dans args(Lx/y/C;->m()) :
                                si v dans clés(H4) :
                                        put(H3,Lx/y/C;->m(),
                                        {get(H4,v),/}
                        si ret(Lx/y/C;->m()) dans clés(H4) :
                                put(H3,Lx/y/C;->m(),
                                {/,get(H4,ret(Lx/y/C;->m())))
                        put(H1,Lx/y/C;->m(),
                        method(Lx/y/C;->m()))
```

Avant l'affichage du graphe d'appels, le type de chaque variable (voir figure 12) ainsi que les types effectifs des arguments et de la valeur de retour de chaque méthode (voir figure 13) sont affichés dans la console.

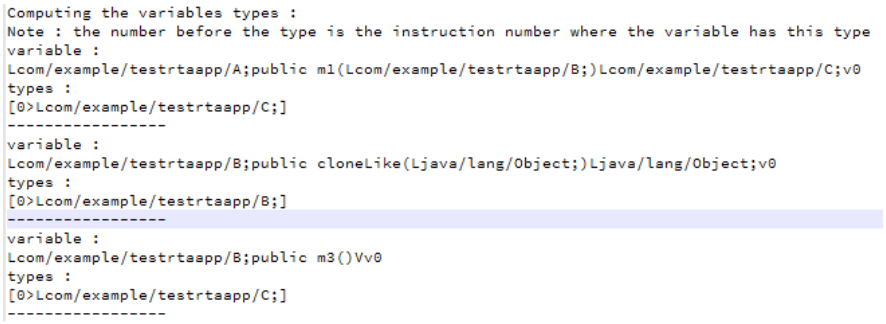

FIGURE 12 – Affichage du type des variables

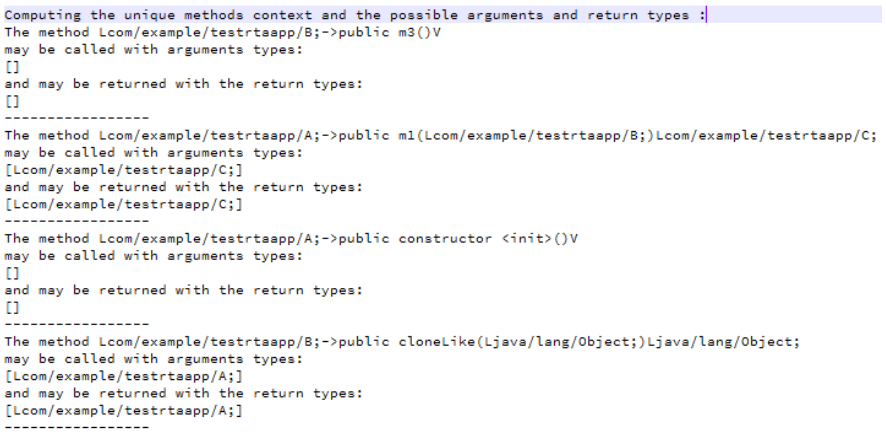

FIGURE 13 – Affichage des possibles types effectifs des arguments et de la valeur de retour des méthodes

4.9 Comparaison pratique des différents algorithmes

Cette Sous-section est destinée à comparer les trois types d'algorithmes de construction de graphes d'appels de l'outil Rundroid amélioré. Cette comparaison sera effectuée selon quatre critères de différenciation :

- le nombre de méthodes
- le nombre d'arêtes
- le temps d'exécution
- la précision de la gestion des caractéristiques Android

4.9.1 Nombre de méthodes

Le tableau ci-dessous établit une comparaison du nombre de méthodes (nombre de n œuds dans le graphe) des trois algorithmes. Dix programmes ont été choisis pour cette comparaison :

- *helloworld* (dans le répertoire test/ HelloWorld-intents de Rundroid)
- Safari
- *Budget* (dans le répertoire testsLaura/testrta6 de Rundroid)

- Google Maps
- Messenger
- Twitter
- *LGame* (dans le répertoire testsLaura/testrta7 de Rundroid)
- Google Earth
- Discord
- Adobe Reader

Programmes	RTA	XTA	0-CFA
helloworld	55	56	55
Safari	628	641	628
Budget	72	73	72
Google Maps	40	57	40
Messenger	115	126	115
Twitter	75	81	75
LGame	119	122	119
Google Earth	40	41	40
Discord	122	143	122
Adobe Reader	142	157	142

TABLE 2 – Comparaison du nombre de méthodes pour RTA, XTA et 0-CFA

On constate que, en moyenne, il y a 9,79 % de méthodes en plus dans le graphe dans l'algorithme XTA que dans les algorithmes RTA et 0-CFA. Cela s'explique par le fait que l'algorithme XTA analyse en plus les méthodes **clinit** des classes et leurs sites d'appels.

Le nombre de méthodes est le même dans les algorithmes RTA et 0-CFA car, dans le contexte de ce travail, c'est le même algorithme qui est utilisé pour construire le graphe d'appels. La différence entre les deux algorithmes est que l'algorithme 0-CFA vérifie, en parallèle, le type de chaque variable.

4.9.2 Nombre d'arêtes

Le tableau ci-dessous établit une comparaison du nombre d'arêtes dans le graphe des trois algorithmes. Dix programmes ont été choisis pour cette comparaison :

- *helloworld* (dans le répertoire test/ HelloWorld-intents de Rundroid)
- Safari
- *Budget* (dans le répertoire testsLaura/testrta6 de Rundroid)
- Google Maps
- Messenger
- Twitter
- *LGame* (dans le répertoire testsLaura/testrta7 de Rundroid)
- Google Earth
- Discord
- Adobe Reader

Programmes	RTA	XTA	0-CFA
helloworld	60	61	60
Safari	1048	1059	1048
Budget	81	82	81
Google Maps	43	60	43
Messenger	128	140	128
Twitter	82	90	82
LGame	191	194	191
Google Earth	43	44	43
Discord	136	161	136
Adode Reader	217	233	217

TABLE 3 – Comparaison du nombre d'arêtes pour RTA, XTA et 0-CFA

On constate qu'il y a, en moyenne, 9,17 % d'arêtes en plus dans le graphe d'appels de XTA que dans les graphes de RTA et 0-CFA. Cela est dû à la même raison qui explique le nombre supérieur de méthodes dans le graphes d'appels de XTA.

4.9.3 Temps d'exécution

Le tableau ci-dessous compare les temps d'exécution en secondes des différents algorithmes (RTA, XTA et 0-CFA). Ce temps d'exécution mesure le temps entre le lancement du programme et l'affichage du graphe d'appels. Dix programmes ont été choisis pour cette comparaison :

- *helloworld* (dans le répertoire test/ HelloWorld-intents de Rundroid)
- Safari
- *Budget* (dans le répertoire testsLaura/testrta6 de Rundroid)
- Google Maps
- Messenger
- Twitter
- *LGame* (dans le répertoire testsLaura/testrta7 de Rundroid)
- Google Earth
- Discord
- Adobe Reader

Programmes	RTA	XTA	0-CFA
helloworld	12	21	14
Safari	250	1188	316
Budget	13	23	16
Google Maps	85	335	86
Messenger	25	44	28
Twitter	63	195	69
LGame	53	2952	297
Google Earth	45	90	45
Discord	85	461	96
Adobe Reader	276	1794	299

TABLE 4 – Comparaison du temps d'exécution (en secondes) pour RTA, XTA et 0-CFA

Nous pouvons constater que la rapport médian entre l'algorithme XTA et RTA est d'environ 7,04 tandis que celui entre l'algorithme 0-CFA et RTA est d'environ

1,15. Nous pouvons également constater que le passage à l'échelle de l'algorithme XTA est mauvais. Par exemple, pour le programme *LGame* l'algorithme XTA est 55,7 fois plus lent que l'algorithme RTA. Cela s'explique par le fait que plusieurs itérations de l'algorithme sont effectuées (jusqu'à ce que les ensembles ne changent plus). De plus, de manière générale, si une classe correspondant à une activité est instanciée, il y a de grandes chances pour qu'au moins une des méthodes appelées par les méthodes de cette classe deviennent atteignable. Or, le cas échéant, la profondeur du graphe augmente car les méthodes appelées appellent elles-mêmes d'autres méthodes (parfois du super-type) qui peuvent appeler d'autres méthodes activités (notamment de super-classes d'activités) et le graphe d'appels peut ainsi devenir très vaste, donc plus lent à construire.

4.9.4 Caractéristiques d' Android

Cette Sous-section compare la précision de la gestion des caractéristiques d'Android des trois algorithmes. Pour chaque caractéristique d'Android, on regarde combien d'éléments de cette caractéristique ont été détectés par l'algorithme. On divise ensuite ce nombre par le nombre effectif d'éléments de la caractéristique. Par exemple, pour les éléments de *layout*, on regarde combien d'éléments de *layout* et de méthodes appelées par la méthode **onClick** implicitement sont détectés par l'algorithme. Autrement dit, on regarde combien d'arêtes se trouvent entre **setContentView** et les constructeurs des éléments de *layout* et combien d'arêtes relient entre la méthode **onClick** et les méthodes appelées par cette dernière. On divise ensuite ce nombre par le nombre effectif d'éléments de *layout* et de méthodes appelées par la méthode **onClick**.

Il n'y a pas de variation de la précision à gérer les caractéristiques d'Android entre les trois algorithmes (en sachant que la méthode de gestion de ces caractéristiques est la même dans chaque algorithme). En effet, les dix programmes sont à 100% pour chacune des caractéristiques Android et pour chaque algorithme.

5 Comparaison de l'outil Rundroid amélioré avec d'autres outils de construction de graphes d'appels

Cette Section est destinée à comparer l'outil Rundroid amélioré et les outils de construction de graphes d'appels existants : WALA, Soot et AndroGuard (voir tableau 5). La comparaison se fera selon trois critères : le temps d'exécution (en secondes), la précision de la gestion des points d'entrées multiples et la précision de la gestion de la réflexion (pour WALA uniquement). Ici, le temps d'exécution pour Soot, WALA et AndroGuard prend uniquement en compte le temps de construction du graphe d'appels (et pas de l'affichage car ils n'affichent pas le graphe après la construction automatiquement). L'algorithme utilisé sera 0-CFA pour Rundroid, VTA pour Soot (le graphe pour 0-CFA de Soot est exactement le même) et 0-CFA pour WALA (AndroGuard ne possède pas un algorithme spécifique de construction de graphes d'appels). Les programmes utilisés sont toujours les programmes dont le code se trouve dans la Section Annexes.

Programmes	Rundroid	Soot	WALA	AndroGuard
Programme 1	Temps : 29 Points d'entrée : 3/3 (100%) Réflexion : 2/2 (100%)	Temps : 136 Points d'entrée : 3/3 (100%) Réflexion : /	Temps : 49 Points d'entrée : 3/3 (100%) Réflexion : 0/2 (0%)	Temps : 92 Points d'entrée : 3/3 (100%) Réflexion : /
Programme 2	Temps : 30 Points d'entrée : 3/3 (100%) Réflexion : 2/2 (100%)	Temps : 120 Points d'entrée : 3/3 (100%) Réflexion : /	Temps : 49 Points d'entrée : 3/3 (100%) Réflexion : 0/2 (0%)	Temps : 110 Points d'entrée : 3/3 (100%) Réflexion : /
Programme 3	Temps : 32 Points d'entrée : 2/2 (100%) Réflexion : 1/1 (100%)	Temps : 131 Points d'entrée : 2/2 (100%) Réflexion : /	Temps : 50 Points d'entrée : 2/2 (100%) Réflexion : 0/1 (0%)	Temps : 124 Points d'entrée : 2/2 (100%) Réflexion : /
Programme 4	Temps : 38 Points d'entrée : 2/2 (100%) Réflexion : 1/1 (100%)	Temps : 126 Points d'entrée : 2/2 (100%) Réflexion : /	Temps : 52 Points d'entrée : 2/2 (100%) Réflexion : 0/1 (0%)	Temps : 107 Points d'entrée : 2/2 (100%) Réflexion : /
Programme 5	Temps : 28 Points d'entrée : 4/4 (100%) Réflexion : 2/2 (100%)	Temps : 122 Points d'entrée : 4/4 (100%) Réflexion : /	Temps : 66 Points d'entrée : 4/4 (100%) Réflexion : 0/2 (0%)	Temps : 94 Points d'entrée : 4/4 (100%) Réflexion : /
Programme 6	Temps : 30 Points d'entrée : 3/3 (100%) Réflexion : 2/2 (100%)	Temps : 121 Points d'entrée : 3/3 (100%) Réflexion : /	Temps : 91 Points d'entrée : 3/3 (100%) Réflexion : 0/2 (0%)	Temps : 95 Points d'entrée : 3/3 (100%) Réflexion : /
Programme 7	Temps : 32 Points d'entrée : 2/2 (100%) Réflexion : 1/1 (100%)	Temps : 137 Points d'entrée : 2/2 (100%) Réflexion : /	Temps : 56 Points d'entrée : 2/2 (100%) Réflexion : 0/1 (0%)	Temps : 94 Points d'entrée : 2/2 (100%) Réflexion : /

TABLE 5 – Comparaison de l'outil Rundroid amélioré et des outils existants

Nous pouvons constater que Rundroid est le plus rapide des outils à cette échelle (utilisé sur de petits programmes). L'hypothèse la plus probable expliquant cette rapidité est que Rundroid ne transforme pas le *bytecode* Dalvik en une représentation intermédiaire ; il le traite tel quel.

Concernant les points d'entrée, tous les outils existants arrivent à détecter la totalité de ceux-ci.

Pour ce qui est de la réflexion, la différence entre Rundroid et WALA est flagrante : WALA n'arrive pas, dans la plupart des cas, à détecter les méthodes appelées par la réflexion. De plus, même si une méthode appelée via cette technique apparaît dans son graphe d'appels, l'outil WALA ne fera pas figurer d'arête entre la méthode **invoke** et la méthode à appeler via la réflexion.

6 Perspectives et conclusion

Même si les algorithmes RTA, XTA et 0-CFA de l'outil Rundroid amélioré

semblent performants sur de petites applications, le passage à l'échelle de ces derniers n'est pas optimal. En particulier, l'algorithme XTA devient très vite lent lorsque la taille de l'application augmente et que des classes qui sont des activités sont instanciées comme expliqué dans la Sous-section 4.9. Par la suite, l'idéal serait de trouver un moyen de le réimplémenter de sorte que son nombre d'itérations soit de l'ordre O(1).

La suite logique de ce travail serait aussi d'implémenter l'algorithme k-CFA qui est une version de l'algorithme 0-CFA où chaque contexte dans lequel une méthode est appelée est retenu ([29]). Cela signifie qu'il serait possible de dire que la méthode d'un contexte particulier est appelée avec certains types d'arguments et de retour et que la même méthode, dans un autre contexte, est appelée avec d'autres types d'arguments et de retour. Dans l'algorithme 0-CFA, les contextes ne sont pas différenciés.

Les graphes d'appels construits à l'aide de ces algorithmes pourraient aussi, par la suite, servir à améliorer la précision de la transformation du *bytecode* en programme logique avec contraintes. En effet, avec un graphe d'appels, il est possible de visualiser plus facilement le flux d'exécution et de voir des appels entre méthodes qui ne seraient pas visibles au premier coup d'œil.

Pour que les algorithmes soient encore plus précis, une autre amélioration possible serait de leur faire également construire des graphes de flux de contrôle interprocéduraux. De cette manière, le flux d'exécution serait enrichi, en plus des appels de méthodes, des autres instructions du *bytecode*.

Un autre grand problème est la gestion des *threads* dans le graphe d'appels, qui, même pour les programmes Java, restent un défi de taille. En incluant les *threads* dans le graphe d'appels, sa précision serait encore augmentée.

L'apport de l'outil Rundroid étendu est un graphe d'appels précis, enrichi des caractéristiques d'Android et offrant le choix de l'algorithme de construction. Inclure ces caractéristiques dans le graphe d'appels permet de découvrir des appels de méthodes implicites dans l'application qu'un graphe sans les *intents*, la réflexion, ... n'aurait pas mis en lumière. Prenons uniquement l'exemple des points d'entrées multiples. Nous pouvons déjà constater que, sans cette caractéristique, un outil ne saurait pas où commencer le graphe car il n'y a pas de méthode **main()** comme en Java dans les applications Android. Les éléments de *layout*, la réflexion et les *intents*, permettent, quant à eux, d'affiner le graphe d'appels. En effet, des appels de méthodes peuvent être cachés à travers ces caractéristiques.

En ce qui concerne les *intents*, si nous prenons l'exemple du lancement d'activité, aucun outil existant ne faisait le lien entre la méthode **startActivity** et les méthodes **onCreate** et **onStart** de l'activité à lancer, même si toutes ces méthodes figuraient dans le graphe d'appels. Or, il existe bel et bien un appel implicite entre ces deux méthodes. En prenant cela en compte, nous pouvons avoir une vision de l'ordre des appels de méthodes plus précise.

Pour la réflexion, le problème est similaire : aucun outil ne faisait le lien entre la méthode **invoke** et la méthode à appeler via la réflexion, même si ces deux méthodes sont dans le graphe d'appels. De plus, il est rare que la méthode à appeler via la réflexion figure dans le graphe à moins qu'elle n'aie déjà été appelée sans la réflexion. Dans la plupart des cas, cette méthode restait donc invisible. Cela a été réglé dans l'outil Rundroid étendu.

Concernant les éléments de *layout*, aucun outil existant ne fait le lien entre la méthode **setContentView** d'une activité et les constructeurs des éléments graphiques du fichier de *layout* de cette activité. Le lien n'est pas fait non plus entre la méthode **onClick** et les méthodes qu'elles appelent implicitement qui sont citées dans le fichier de *layout*.

Pour résumer les deux paragraphes précédents, les éléments de *layout* et la réflexion ajoute des informations cruciales pour l'ordre d'appel des méthodes et ajoute même des méthodes qui ne seraient pas détectées en temps normal mais qui sont bel et bien appelées.

Pour entrer plus dans le détail, comme nous avons pu le constater, Soot, WALA et AndroGuard n'incluent pas les caractéristiques d'Android dans leur graphe d'appels. Excepté pour les points d'entrées multiples qu'ils détectent tous les trois avec une grande précision. Même si la gestion de la réflexion est possible dans WALA, la plupart du temps, une des seules méthodes que cet outil trouve en rapport avec la réflexion est **forName**. De plus, même si la méthode **invoke** est trouvée, le lien n'est pas fait entre cette dernière et son argument (la méthode à appeler via la réflexion). Malgré la précision du graphe d'appels de AndroGuard, dû à son grand nombre d'arêtes et de méthodes, les appels implicites générés par les éléments de *layout*, la réflexion ou les *intents* ne figurent pas dans le graphe. Une partie du flux d'exécution est donc manquant comme expliqué plus haut.

Nous avons également pu constater dans la Section 3 que les graphes d'appels des applications Android de Soot, peu importe l'algorithme, sont exactement les mêmes. Ce qui fait perdre l'outil en richesse, vu la variété d'algorithmes proposée. Dans l'outil Rundroid étendu, les graphes d'appels de l'algorithme RTA et 0-CFA sont également identiques. Cependant, la différence se situe dans l'affichage de la console. En effet, l'algorithme 0-CFA, avant d'afficher le graphe d'appels, affiche le type de chaque variable et les types effectifs des arguments et de retour possibles de chaque méthode. De même pour l'algorithme XTA : même si son graphe est très similaire à celui de l'algorithme RTA, la grande différence est que les ensembles des méthodes et des champs sont affichés dans la console.

Pour conclure, l'outil Rundroid étendu dans ce travail permet désormais d'obtenir un flux d'appels quasi complet (en ne prenant pas en compte les appels entre librairies ni les *threads*), ce qui comble une lacune des outils de construction de graphes d'appels existants.

7 Annexes

7.1 Code du programme Test

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

    public void clickMe() {
        Intent implintent = new Intent();
        implintent.setAction(Intent.ACTION_MAIN);
        implintent.addCategory(Intent.CATEGORY_LAUNCHER);
        implintent.setData(Uri.parse("data:newdata"));
        startActivity(implintent);
    }

}

public class SecondMainActivity  extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

}

public class A  {

    public void reflectMethod(){ }

}
```

Listing 1 – Code du programme 1

7.2 Code du programme 1

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        A activityClass = new A();
```

```
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
            Method method2 = activity.getDeclaredMethod("reflectMethod2");
            method2.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

    public static void clickMe() {
        System.out.println("clicked");
    }
}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Intent implintent = new Intent();
        implintent.setAction(Intent.ACTION_MAIN);
        Uri data = Uri.parse("data :newdata");
        implintent.setData(data);
        startActivity(implintent);
    }
}

public class ThirdMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

public class A {

    public void reflectMethod(){
        MainActivity.clickMe();
    }

    public void reflectMethod2(){
        MainActivity.clickMe();
    }
}
```

Listing 2 – Code du programme 1

7.3 Code du programme 2

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

    public void clickMe(){
```

```
        System.out.println("clicked");
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
    }

    public static void m2(){
        ThirdMainActivity.m3();
    }
}

public class ThirdMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public static void m3(){
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod3");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
    }
}

public class A  {

    public void reflectMethod(){
        m();
    }

    public void reflectMethod3(){}

    public void m() {
        SecondMainActivity.m2();
    }
}
```

Listing 3 – Code du programme 2

7.4 Code du programme 3

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
            Method method2 = activity.getDeclaredMethod("reflectMethod2");
            method2.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,TestService.class);
        startService(explintent);
    }

    public void clickMe() {
        System.out.println("clicked");
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "second activity created", Toast.LENGTH_SHORT).show();
    }
}

public class A  {

    public void reflectMethod(){
        m();
    }

    public void reflectMethod2(){
        m();
    }

    public void m() {}
}

public class TestService extends Service {

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "service started", Toast.LENGTH_SHORT).show();
        Intent explintent2 = new Intent(this,SecondMainActivity.class);
        startActivity(explintent2);
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

Listing 4 – Code du programme 3

7.5 Code du programme 4

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,TestService.class);
        startService(explintent);
    }

    public void clickMe() {
        System.out.println("clicked");
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "second activity created", Toast.LENGTH_SHORT).show();
        Intent implintent = new Intent();
        implintent.setAction(Intent.ACTION_VIEW);
        Uri data2 = Uri.parse("data:newdata2");
        implintent.setData(data2);
        sendBroadcast(implintent);
    }
}

public class TestService extends Service {

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "service started", Toast.LENGTH_SHORT).show();
        Intent explintent2 = new Intent(this,SecondMainActivity.class);
        startActivity(explintent2);
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}

public class TestBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i("receiver","data receiver");
    }
}
```

Listing 5 – Code du programme 4

7.6 Code du programme 5

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

    public void clickMe() {
        System.out.println("clicked");
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "second activity created", Toast.LENGTH_SHORT).show();
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod2");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
    }
}
public class ThirdMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "third activity created", Toast.LENGTH_SHORT).show();
    }
}

public class FourthMainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "fourth activity created", Toast.LENGTH_SHORT).show();
    }
}
```

Listing 6 – Code du programme 5

7.7 Code du programme 6

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        A activityClass = new A();
```

```
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

    public void clickMe() {
        System.out.println("clicked");
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.second_activity_main);
        Toast.makeText(this, "second activity created", Toast.LENGTH_SHORT).show();
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod2");
            method.invoke(activityClass,null);
        } catch (Exception e){ e.printStackTrace(); }
    }

    public void secondClick(){
        Intent intent = new Intent(this,ThirdMainActivity.class);
        startActivity(intent);
    }
}

public class ThirdMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "third activity created", Toast.LENGTH_SHORT).show();
    }
}
```

Listing 7 – Code du programme 6

7.8 Code du programme 7

```
package com.example.test1;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void clickMe() {
        A activityClass = new A();
        try {
            Class activity = Class.forName("com.example.test1.A");
            Method method = activity.getDeclaredMethod("reflectMethod");
            method.invoke(activityClass,null);
```

```
        } catch (Exception e){ e.printStackTrace(); }
        Intent explintent = new Intent(this,SecondMainActivity.class);
        startActivity(explintent);
    }

}

public class SecondMainActivity  extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText(this, "second activity created", Toast.LENGTH_SHORT).show();
    }
}

public class A  {

    public void reflectMethod(){
        m();
    }

    public void m() {}
}
```

Listing 8 – Code du programme 7

Bibliographie

[1] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 210–221. ACM, 2018.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid : precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014.

[3] Bhargav Nagaraja Bhatt and Carlo A. Furia. Automated repair of resource leaks in android applications. *CoRR*, abs/2003.03201, 2020.

[4] Malinda Dilhara, Haipeng Cai, and John Jenkins. Automated detection and repair of incompatible uses of runtime permissions in android apps. In Christine Julien, Grace A. Lewis, and Itai Segall, editors, *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, pages 67–71. ACM, 2018.

[5] Ehsan Edalat, Babak Sadeghiyan, and Fatemeh Ghassemi. Considroid : A concolic-based tool for detecting SQL injection vulnerability in android apps. *CoRR*, abs/1811.10448, 2018.

[6] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In Ahmad-Reza Sadeghi, Blaine Nelson, Christos Dimitrakakis, and Elaine Shi, editors, *AISec'13, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, pages 45–54. ACM, 2013.

[7] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 167–177. ACM, 2018.

[8] http ://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.

[9] https ://androguard.readthedocs.io/en/latest/, 2012.

[10] https ://code.google.com/archive/p/android4me/downloads.

[11] https ://developer.android.com/guide/components/intents-filters.

[12] https ://developer.android.com/guide/topics/manifest/manifest-intro.

[13] https ://gephi.org/, 2008.

[14] https ://github.com/Sable/soot, 2012.

[15] https ://github.com/secure-software engineering/soot-infoflow-android.

[16] https ://ibotpeaches.github.io/Apktool/, 2010.

[17] https ://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[18] https ://www.graphviz.org/, 2001.

[19] https ://www.vogella.com/tutorials/AndroidBroadcastReceiver/article.html.

[20] http ://wala.sourceforge.net/wiki/index.php/Main_Page, 2006.

[21] Yongjian Hu and Iulian Neamtiu. Static detection of event-based races in android apps. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 257–270. ACM, 2018.

[22] Sungho Lee and Sukyoung Ryu. Adlib : analyzer for mobile ad platform libraries. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 262–272. ACM, 2019.

[23] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps : A systematic literature review. *Inf. Softw. Technol.*, 88 :67–95, 2017.

[24] Shuyuan Liu and Gang Gan. Graph structure-based clustering algorithm for android third-party libraries. *IOP Conference Series : Earth and Environmental Science*, 428 :012009, jan 2020.

[25] Yu Liu, Kai Guo, Xiangdong Huang, Zhou Zhou, and Yichi Zhang. Detecting android malwares with high-efficient hybrid analyzing methods. *Mobile Information Systems*, 2018 :1649703 :1–1649703 :12, 2018.

[26] Omid Mirzaei, Guillermo Suarez-Tangil, José María de Fuentes, Juan Tapiador, and Gianluca Stringhini. Andrensemble : Leveraging API ensembles to characterize android malware families. In Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang, editors, *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 307–314. ACM, 2019.

[27] Bailey Brian Nottingham. An algorithm and implementation to detect covert channels and data leakage in mobile applications, 2019.

[28] Pooja Patil. FAST COMMUNITY STRUCTURE ANALYSIS OF CALL GRAPHS FOR MALWARE DETECTION. 5 2019.

[29] Jean Privat. Compilation de langage à objets analyse de types et graphe dappels en compilation séparée, 2002.

[30] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 474–486. ACM, 2016.

[31] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Mary Beth Rosson and Doug Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, pages 281–293. ACM, 2000.

[32] Wei Wang, Zhenzhen Gao, Meichen Zhao, Yidong Li, Jiqiang Liu, and Xiangliang Zhang. Droidensemble : Detecting android malicious applications with ensemble of string and structural static features. *IEEE Access*, 6 :31798–31807, 2018.

[33] Yifei Zhang, Yue Li, Tian Tan, and Jingling Xue. Ripple : Reflection analysis for android apps in incomplete information environments. *Software : Practice and Experience*, 48(8) :1419–1437, 2018.

[34] Mingsong Zhou, Fanping Zeng, Yu Zhang, Chengcheng Lv, Zhao Chen, and Guozhu Chen. Automatic generation of capability leaks' exploits for android applications. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, pages 291–295. IEEE, 2019.